Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems

by

Hakim Weatherspoon

B.S. (University of Washington, Seattle) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor John Kubiatowicz, Chair
Professor Anthony Joseph
Professor John Chuang

Fall 2006

The dissertation of Hakim Weatherspoon is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems

Abstract

Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems

by

Hakim Weatherspoon
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Kubiatowicz, Chair

As the amount of digital assets increase, systems that ensure the durability, integrity, and accessibility of digital data become increasingly important. Distributed on-line archival storage systems are designed for this very purpose. This thesis explores several important challenges pertaining to fault tolerance, repair, and integrity that must be addressed to build such systems.

The first part of this thesis explores how to maintain durability via fault tolerance and repair and presents many insights on how to do so efficiently. Fault tolerance ensures that data is not lost due to server failure. Replication is the canonical solution for data fault tolerance. The challenge is knowing how many replicas to create and where to store them. Fault tolerance alone, however, is not sufficient to prevent data loss as the last replica will eventually fail. Thus, repair is required to replace replicas lost to failure. The system must monitor and detect server failure and create replicas in response. The problem is that not all server failure results in loss of data and the system can be tricked into creating replicas unnecessarily. The challenge is knowing when to create replicas. Both fault tolerance and repair are required to prevent the last replica from being lost, hence, maintain data durability.

The second part of this thesis explores how to ensure the integrity of data. Integrity ensures that the state of data stored in the system always reflects changes made by the owner. It includes non-repudiably binding owner to data and ensuring that only the owner can modify data, returned data is the same as stored, and the last write is returned in subsequent reads. The challenge is efficiency since requiring cryptography and consistency in the wide-area can easily be prohibitive.

Next, we exploit a secure log to efficiently ensure integrity. We demonstrate how the narrow interface of a secure, append-only log simplifies the design of distributed wide-area storage

systems. The system inherits the security and integrity properties of the log. We describe how to replicate the log for increased durability while ensuring consistency among the replicas. We present a repair algorithm that maintains sufficient replication levels as machines fail. Finally, the design uses aggregation to improve efficiency. Although simple, this interface is powerful enough to implement a variety of interesting applications.

Finally, we apply the insights and architecture to a prototype called Antiquity. Antiquity efficiently maintains the durability and integrity of data. It has been running in the wide area on 400+ PlanetLab servers where we maintain the consistency, durability, and integrity of nearly 20,000 logs totaling more than 84 GB of data despite the constant churn of servers (a quarter of the servers experience a failure every hour).

Professor John Kubiatowicz
Dissertation Committee Chair

Contents

List of Figures **vi**

List of Tables **xiv**

I Defining Scope of Problem **1**

1 Introduction **2**
1.1 Overview . . . 3
1.1.1 Maintaining Durability . . . 3
1.1.2 Maintaining Integrity . . . 6
1.1.3 Putting It All Together . . . 9
1.2 Challenges . . . 10
1.2.1 Assumptions . . . 10
1.2.2 The Fault Tolerance Problem . . . 11
1.2.3 The Repair Problem . . . 12
1.2.4 The Integrity Problem . . . 14
1.3 Architecture for a Solution: Antiquity Prototype . . . 17
1.4 Historical Perspective . . . 18
1.5 Lessons Learned . . . 19
1.6 Contributions . . . 21
1.7 Summary . . . 22

2 Methodology **23**
2.1 Failure Characteristics . . . 24
2.1.1 Analyzing the Behavior of PlanetLab . . . 25
2.1.2 PlanetLab First Interval: Insights into Correlated Failures . . . 27
2.1.3 PlanetLab Second Interval: Insights into Matured System and Operation . . . 29
2.1.4 Synthetic trace . . . 30
2.2 Analyzing Algorithmic Solutions . . . 31
2.2.1 Algorithmic Solution Representation . . . 32
2.2.2 Metrics . . . 33
2.2.3 Statistics Gathering and Analysis . . . 33
2.2.4 Algorithmic Solution Comparison Criteria . . . 34

2.2.5 Environments . . . 35
2.3 Discussion . . . 36

II Maintaining Data Durability Through Fault Tolerance and Repair 38

3 Fault Tolerance and Repair Overview 39
3.1 System Model . . . 41
3.2 Example . . . 41

4 Fault Tolerance 43
4.1 Choosing Redundancy Type . . . 43
4.1.1 Erasure-coding versus Replication . . . 45
4.1.2 Complexity of Erasure-Codes and Self-Verifying Data . . . 53
4.2 Choosing the Number of Replicas to Create . . . 56
4.2.1 System Model . . . 56
4.2.2 Generating a Markov Model . . . 60
4.2.3 Creation versus Failure Rate . . . 61
4.2.4 Choosing r_L . . . 64
4.3 Choosing Where to Store Replicas . . . 68
4.3.1 Increasing Durability Through Repair Parallelism with Scope . . . 70
4.3.2 Placement Strategies, Failure Predictors, and Durability . . . 74
4.4 Summary . . . 79

5 Repair 82
5.1 Reducing Transient Costs with Monitoring and Timeout-based Failure Detectors . . . 83
5.1.1 Failure detectors . . . 84
5.1.2 Evaluation of Timeout-based Failure Detectors . . . 85
5.2 Reducing Transient Costs with Extra Replication . . . 86
5.2.1 Estimator Algorithm . . . 88
5.2.2 Evaluation of Extra Replication . . . 92
5.3 Reducing Transient Costs with Reintegration . . . 94
5.3.1 Carbonite details . . . 95
5.3.2 Reintegration reduces maintenance . . . 96
5.3.3 How many replicas? . . . 97
5.3.4 Create replicas as needed . . . 99
5.3.5 Reintegration and Erasure-coding . . . 101
5.4 Summary . . . 102

III Exploiting a Secure Log for Wide-Area Distributed Storage 103

6 Secure Log Overview 104
6.1 Overview . . . 106
6.1.1 Storage System Goals . . . 106

6.1.2 System Model . . . 107
6.1.3 Assumptions . . . 108
6.2 Secure Log Details . . . 109
6.3 Semantics of a Distributed Secure Log . . . 112
6.4 Example uses of a Secure Log . . . 114
6.4.1 File System Interface . . . 114
6.4.2 Database Example . . . 116
6.4.3 Tamper-resistant syslog . . . 116

7 The Secure Log Interface 117
7.1 Background and Prior Work . . . 118
7.1.1 Self-verifying Data . . . 118
7.1.2 Distributed hash table (DHT) storage systems . . . 119
7.1.3 Prior Aggregation Systems . . . 121
7.2 How to use an Aggregation Interface to Construct a Secure Log . . . 121
7.2.1 Constructing a Secure Log . . . 122
7.2.2 Reading data from a Secure Log . . . 124
7.2.3 Other benefits of Aggregation . . . 125
7.3 A Distributed Secure Log and Error Handling . . . 125
7.4 Discussion . . . 126

8 Dynamic Byzantine Quorums for Consistency and Durability 128
8.1 Background and Prior Work . . . 129
8.2 Protocol Requirements . . . 131
8.3 Protocol Assumptions . . . 132
8.4 Quorum Repair . . . 133
8.4.1 Challenges . . . 133
8.4.2 Triggering repair . . . 136
8.4.3 Initializing a configuration after repair . . . 137
8.4.4 Certificates and Soundness Proofs . . . 138
8.4.5 The Repair Audit Protocol . . . 139
8.5 Protocol Details . . . 139
8.5.1 Base Write Protocol . . . 140
8.5.2 The `create()` Protocol . . . 141
8.5.3 The `append()` Protocol . . . 142
8.5.4 The `repair()` Protocol . . . 144
8.5.5 Transient Server Failure Protocol . . . 145
8.6 Protocol Correctness . . . 146
8.6.1 Protocol Consistency . . . 146
8.6.2 Protocol Durability . . . 146
8.6.3 Protocol Liveness . . . 147
8.7 Discussion . . . 147

9 Utilizing Distributed Hash Table (DHT) Technology for Data Maintenance 149
9.1 Publishing and Locating Extent Replicas . . . 150
9.2 Monitoring Server Availability and Triggering Repair Audits . . . 151
9.3 Discussion and Limitations . . . 153

IV Antiquity: Prototype Implementation and Evaluation 155

10 Antiquity 156
10.1 Architecture Overview . . . 157
10.2 Gateways, Coordinators, Distributed Hash Tables, and Protocol Details . . . 159
10.2.1 Path of a `create()`, `append()`, and `repair()` . . . 160
10.2.2 Breakdown of latencies for all operations . . . 162
10.3 Summary and Discussion . . . 164

11 Evaluation 165
11.1 Experimental Environment . . . 165
11.2 Cluster Deployment . . . 166
11.3 PlanetLab Deployment . . . 170
11.3.1 Quorum Consistency and Availability . . . 172
11.3.2 Quorum Repair . . . 173
11.4 A Versioning Back-up Application . . . 174
11.5 Experience and Discussion . . . 174

V Related and Future Work 178

12 Related Work 179
12.1 Logs . . . 179
12.2 Byzantine Fault-Tolerant Services . . . 179
12.3 Wide-area Distributed Storage Systems . . . 180
12.4 Replicated Systems . . . 180
12.5 Replication analysis . . . 181
12.6 Replicated systems . . . 182
12.7 Digital Libraries . . . 183

13 Future Work 184
13.1 Proactive Replication for Data Durability . . . 184
13.2 Closed-loop, Proactive Repair, for Data Durability . . . 186
13.3 Administrator Discussion . . . 187
13.4 OceanStore as an Application . . . 187
13.5 Summary . . . 188

14 Concluding Remarks 189

Bibliography 191

A Durability Derivation **202**

B Glossary of Terms **204**

List of Figures

1.1 Example Maintaining Durability in a Distributed Wide-area On-line Archival Storage System. (a) fault tolerance defines the system's ability to tolerate server failure without loss of data. It includes choosing the type of redundancy (e.g. replication or erasure-coding), number of replicas, and where to store replicas. (b) Repair is the process of replacing replicas lost to server failure (such as Louisiana). It includes detecting server failure and creating new replicas (such as Georgia). 4
1.2 Example Maintaining Integrity in a Distributed Wide-area On-line Archival Storage System. (a) A data object with the value *A* is replicated onto four servers via a request by a workstation. In (b), the same workstation attempts to add value *B*, predicated on *A* already being stored. The request succeeds since it reaches a threshold of servers (Illinois, Louisiana, and New York). The request did not reach the Georgia server, however, possibly due to network transmission error or transient failure. In (c), a laptop, which possesses the same private key as the workstation, attempts to add value *C*, predicated on *A* already being stored. The request fails since the predicate fails on a threshold of servers. Note that the server in Georgia applies *C* since the predicate matches local state. However, the system should return value *B* in any subsequent reads. 7
1.3 Example Maintaining Durability and Integrity Together in a Distributed Wide-area On-line Archival Storage System. In (a) and (b), a server storing a replica in Louisiana permanently fails. In (c), during repair, the system should initialize the new configuration to the state reflecting the latest successful write, *A B*. 9

2.1 Example Server Failure Trace. 24
2.2 Server Attrition of All-pairs Ping Data Set. 26
2.3 PlanetLab first Interval Characteristics (All-pairs Ping). Sessiontime, downtimes, and availability distributions. 28
2.4 PlanetLab First Interval Trace Characteristics (All-pairs Ping). Permanent server failure interarrival distribution. 29
2.5 PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Sessiontime, downtime, and availability distributions. 30
2.6 PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Permanent server failure interarrival distribution. 31

3.1 Example of Maintaining Data in a Wide Area Storage System. 42

4.1 Fraction of Blocks Lost Per Year (FBLPY) for a rate $\frac{1}{4}$ erasure-encoded block. Disks fail after five years and a repair process reconstructs data regularly. The four-fragment case (top line) is equivalent to simple replication on four servers. Increasing the number of fragments increases the durability of a block while total storage overhead remains constant. Notice, for example, that for a repair interval of 6 months the four-way replication (top line) loses 0.03 (3%) blocks per year while the 64 fragments, any 16 of which are sufficient to reconstruct (bottom line) loses 10^{-35} blocks per year. 44
4.2 Disk Mortality Distribution [PH02]. 46
4.3 Hybrid Update Architecture: Updates are sent to a central "Archiver", which produces archival fragments at the same time that it updates live replicas. Clients can achieve low-latency read access by utilizing replicas directly. 52
4.4 (a) Verification Tree: is a hierarchical hash over the fragments and data of a block. The top-most hash is the block's *GUID*. (b) Verification Fragments: hashes required to verify the integrity of a particular fragment. 55
4.5 Example repair process. The replica location and repair service coordinates repair process: new servers prioritize downloading new replicas. Initially, there are four servers and four objects (A thru D) with $r_L = 3$ for each object. (a) Server 1 fails. The replica location and repair service selects a new server, server 5, to download lost replicas. (b) Before any repair completes, server 2 fails. The replica location and repair service selects server 6 to download lost replicas and communicates new download priority to server 5 (A,D,C instead of A,C,D since D has less replicas that exist than C). (c) All repair completes. Notice that object replica C on server 5 and B on server 6 waited for an entire servers worth of repairs to complete before they completed. 57
4.6 Example number of replicas that exist over time. Initially, $r_L = 8$ replicas of an object are inserted into the system. At time 3, a server storing a replica fails. The failure is detected at time 4 and repaired by time 6. The repair lag is due to constrained resources such as access link bandwidth that restrict the number of objects that a server can repair in a given time period. Furthermore, the newly triggered repair would have to wait for previously triggered repairs to complete. Later, another server fails at time 9. But before repair can complete, another server fails at time 11 bringing the number of replicas that exist down to 6. The lost replicas are replaced by time 14. The lowest number of replicas that exist is 5 at time 21. 58
4.7 A continuous time Markov model for the process of replica failure and repair for a system that maintains three replicas ($r_L = 5$). Numbered states correspond to the number of replicas of each object that exist. Transitions to the left occur at the rate at which replicas are lost; right-moving transitions happen at the replica creation rate. 60

4.8 Average number of replicas at the end of a two year synthetic trace for varying values of θ. This Figures represents a set of simulations where we reduced the bandwidth per server (x-axis) effectively reducing the replica creation rate μ (and θ). The input to the simulator was a synthetic failure trace with a 632 servers and a server failure rate of λ_f = 1 per year. The storage load maintained was 1TB of unique data (50,000 20MB objects). As a result, the total replicated data was 2TB, 4TB, 6TB, and 8TB for r_L = 2, 4, 6, 8, respectively. Finally, each experiment was run with a specific available bandwidth per server that ranged from 100 B/s to 1,200 B/s. 63
4.9 Frequency of "simultaneous" failures in the PlanetLab trace. These counts are derived from breaking the trace into non-overlapping 24 and 72 hour periods and noting the number of permanent failures that occur in each period. If there are *x* replicas of an object, there were *y* chances in the trace for the object to be lost; this would happen if the remaining replicas were not able to respond quickly enough to create new replicas of the object. 65
4.10 Analytic results for the probability of data loss over time. These curves are the solution to the system of differential equations governing a continuous time Markov process that models a replication system running on PlanetLab storing 500GB. At time zero, the system is in state 3 (three replicas) with probability 1.0 (dot-dash line is at 1.0). As time progresses, the system is overwhelmingly likely to be in state 0 which corresponds to object loss (that probability is shown as a bold line in the above plot). 67
4.11 Analytic prediction for object durability after four years on PlanetLab. The *x*-axis shows the initial number of replicas for each object: as the number of replicas is increased, object durability also increases. Each curve plots a different per-server storage load; as load increases, it takes longer to copy objects after a failure and it is more likely that objects will be lost due to simultaneous failures. 68
4.12 Scope. Each unique server *i* has a unique set of servers it monitors and can potentially hold copies of the objects that *i* is responsible for (server set$_i$ $\not\subset$ storage server set$_j$, $\forall i \neq j$). The *size* of that set is the server's *scope*. (a) scope=3 and (b) scope=5. In terms of placement choices, assuming that r_L =3 and object replicas are stored on server *i*'s server set, then there is no choice for (a) and $\binom{\text{scope}}{r_L} = \binom{5}{3}$ choices in (b). . 70
4.13 Example parallel repair with a large scope. Scope is 7 and r_L = 3. Only servers within scope are monitored and there are $\binom{7}{3}$ possible replica sets. The replica location and repair service coordinates the repair process utilizing as many source and destination server pairs as possible. Initially, there are seven servers and seven objects (A thru G) with $r_L = 3$ for each object. (a) Server 1 fails. The replica location and repair service selects as many source and destination server pairs to reduce the repair time. Server 2 downloads replica A from server 3. Similarly, server 5 downloads replica D from server 4 and server 6 downloads replica B from server 7. (b) All repair completes. 72

4.14 Durability for different scopes. Assuming random placement. We vary the target replication level r_L and scope (x-axis). To reduce θ, we limit the bandwidth per server to 1000 B/s in this experiment. Durability is measured via simulation using a two year synthetic trace. Increasing the scope from 5 to 25 servers reduces the fraction of lost objects by an order of magnitude, independent of r_L. 73
4.15 Temporally Correlated Failures. We use a two-dimensional space of conditional downtime probabilities, both p(x is down | y is down) and p(y is down | x is down). Servers x and y are temporally correlated if *both* probabilities are greater than a threshold such as 0.5 or 0.8. (a) upper right quadrant of 2D Correlation. (b) Fraction of Correlated Servers. 22% of the time that a server goes down, there is at least a 50% chance another server in the same site will go down as well. Alternatively, the servers in different sites were not temporally correlated. 75
4.16 Per Server Total Downtime (log scale). 76
4.17 Temporally Correlated Failures with servers with total downtimes longer then 1000 hours removed from consideration. (a) 2D Correlation and (b) Fraction of Correlated Servers (servers w/ total downtime $\leq$ 1000 hours). 33% of the time that a server goes down, who's total downtime is less than 1000 hours, there is at least a 50% chance another server in the same site will go down. The temporally correlated probability increased when we removed the long downtime servers because the number of servers temporally correlated remained relatively unchanged from Figure 4.15 while the total number of servers was reduced by 188. Alternatively, the servers in different sites were not temporally correlated. 77

5.1 The impact of timeouts on bandwidth and durability on a synthetic trace. Figure 5.1(a) shows the number of copies created for various timeout values; (b) shows the corresponding object durability. In this trace, the expected downtime is about 29 hours. Longer timeouts allow the system to mask more transient failures and thus reduce maintenance cost; however, they also reduce durability. 85
5.2 Transient and Permanent Failures over Time. 86
5.3 Example. Cost per Server of Maintaining Data in a Wide Area Storage System. . . 87
5.4 Extra Replication Estimator for Storage Systems on PlanetLab. 92
5.5 Extra Replication. Figures (a), (b), and (c) use the DHT-based storage system like Dhash and Figures (d), (e), and (f) use a directory-based storage system with a Random placement. Figures (a) and (d) shows the number of repairs triggered per week over the course of the trace. Figures (b) and (e) show the average bandwidth per server (averaged over a week) over the course of the trace. Finally, Figures (c) and (f) show the average bandwidth per server as we vary the number of extra replicas and timeout values. 93
5.6 Cost Breakdown for Maintaining Minimum Data Availability for 2 TB of unique data. (a) and (b) Cost breakdown with a unique write rate of 1Kbps and 10 Kbps per server, respectively. Both (a) and (b) fix the data placement strategy to `Random` and timeout $\tau = 1$hr. The cost due to heartbeats is not shown since it was less than 1Kbps. 94

5.7 Each server maintains a list of objects for which it is responsible and monitors the replication level of each object using some synchronization mechanism. In this code, this state is stored in the replicas hash table though an implementation may choose to store it on disk. This code is called periodically to enqueue repairs on those objects that have too few replicas available; the application can issue these requests at its convenience. 95
5.8 A comparison of the total amount of work done by different maintenance algorithms with $r_L = 3$ using a PlanetLab trace (left) and a synthetic trace (right). In all cases, no objects are lost. However, $r_L = 2$ is insufficient: for the PlanetLab trace, even a system that could distinguish permanent from transient failures would lose several objects. 96
5.9 Additional redundancy must be created when the amount of live redundancy drops below the desired amount (3 replicas in this example). The probability of this happening depends solely on the average server availability a and the amount of durable redundancy. This graph shows the probability of a repair action as a function of the amount of durable redundancy, with $a = 0.5$, $a = 0.7$ and $a = 0.9$ for a replication system. 98
5.10 Total repair cost with extra replicas, and with and without reintegration after repair. Without reintegration, extra replicas reduce the rate at which repair is triggered and thus reduce maintenance cost; there is an optimal setting (here $e = 8$). With reintegration, the cost is lowest if no extra replicas are used. 100
5.11 Total repair cost with a rate $r = \frac{m}{n} = \frac{7}{14}$ erasure-coding scheme, reintegration, extra fragments, and with and without replica caching after reconstruction and repair. Without caching, extra fragments reduce the rate at which repair is triggered and thus reduce maintenance cost; there is an optimal setting (here $e = 12$). With caching, the cost is lowest if few extra fragments are used ($e = 0$ to 2). 101

6.1 A log-structured storage infrastructure can provide storage for end-user clients, client-server systems, or replicated services. Each log is identified by a key pair. . . 107
6.2 To compute the verifier for a log, the system uses the recurrence relation $V_i = H(V_{i-1} + H(D_i))$. $V_{-1} = H(PK)$ where PK is a public key. 109
6.3 Semantics of a Distributed Secure Log. (a) A secure log with the value A is initially replicated onto seven servers. In (b), a workstation attempts to `append()` the value B, predicated on A already being stored. The result of the request is sound since it reaches a threshold of servers (servers 3-7). In (c), a laptop, which possesses the same private key as the workstation, simultaneously attempts to `append()` value C, predicated on A already being stored. The result of the request is unsound since the predicate fails on a threshold of servers. Note that the two servers (server 1-2) apply C since the predicate matches local state. However, the system should return value B in any subsequent reads. 113
6.4 (a) An abstract representation of a versioning file system with two versions. A version can reference newly created data and data in previous versions. (V = version, R = root directory, I = file inode, B = data block) (b) An application can write the file system to a log by traversing the tree in a depth-first manner. 115
6.5 A simple file system used as a running example. Map symbols to concrete file system. 116

7.1 Clients divide data into small blocks that are combined into Merkle trees. A key-verified block points to the root of the structure. To update an object, a client overwrites the key-verified block to point to the new root. (V = version, R = version root, I = indirect node, B = data block) 120
7.2 This example illustrates how the client library uses the extended API to write the first version of the file system shown in Figure 6.4. The shaded extent is the mutable log head; immutable extents are shown in white. 124

8.1 Example `create()` request using a Byzantine fault-tolerant quorum. (a) A client attempts to create a secure log with a configuration that includes seven servers and can tolerate two faulty servers ($f = 2$ and $n = 7 > 3f$). After an administrator selects a configuration, the client submits the `create()` request to all the servers in the configuration. (b) The `create()` request succeeds after the client receives positive acknowledgment from a quorum of servers ($q = 7 - 2 = 5$). 130
8.2 Example `create()` request using a Byzantine fault-tolerant agreement. (a) and (c) are similar to the Byzantine quorum `create()` request and acknowledgment in Figure 8.1.(a) and (b), respectively. However, Figure (b) above illustrates that Byzantine agreement protocols use $O(n^2)$ messages over multiple rounds, whereas Byzantine quorums use $O(n)$ over two rounds where the second round is often piggybacked onto subsequent operations [AGG+05]. 131
8.3 A write is successful after a client receives positive acknowledgment from a quorum of q storage servers. Two clients simultaneously submit conflicting writes. During repair, the system should initialize the new configuration to the state reflecting the latest successful write. In these two examples, the server state that can be observed from the clients at time $t = 3$ is the same, but the latest successful write differs. In (a), the client that wrote c received a quorum of positive server acknowledgments and, thus, is successful. In (b), the client that wrote c did not receive a quorum of positive server acknowledgments so the write failed, thus, the new configuration must be initialized to a. 134
8.4 Example total order of sound operations. 137
8.5 Latest soundness proof. From Figure 8.3(a), at time $t = 2$, the latest sound write was c. Assume a quorum of servers (servers 1-7) acknowledged receiving the latest soundness proof (configuration parameters $n = 9$, $q = 7$, $r = 5$, and $f = 2$). This figure shows the administrator's view of the storage system at time $t = 3$ after receiving replies from five servers (servers 1, 2, 5, 8, 9). Assume servers 1 and 2 are malevolent and can either send the latest or old proof and servers 8 and 9 are out-of-date and did not receive the last proof. At least one server response out of five (server 5) contains the latest soundness proof (c). 138

8.6 Local server state for log head and hash-verified extents. It includes proven state (with soundness proof) and pending state (without soundness proof). Proven state includes the latest soundness proof, block_names, and data. Mapping is used to connect extents into a secure log. Proven state is null when an extent is first created, when `create()`, `snapshot()`, or `put()` are pending; otherwise, it is not null. Pending data includes a pending soundness proof (certificate and configuration without server signatures), block_names and data. Pending_map is used by `truncate()`, pending_map points to the extent created during `snapshot()`. Pending state is null if no requests are pending. When a pending request gathers proof of soundness the pending_proof field replaces the proof. `create()`, `snapshot()`, and `put()` replace block_names, data_blocks, and mapping with the associated pending fields. `append()` adds the pending_block_names and pending_data_blocks to block_names and data_blocks fields, respectively. `truncate()`, however, removes block_names and data_blocks fields; additionally, it replaces the mapping field with the pending_map field. 140

8.7 (a) To complete a `create()` request, a client must first request a new configuration from the administrator. The client then sends the configuration along with the signed certificate to storage servers listed in the configuration. (b) To complete an `append()` request, the client must only send a message to each storage server in the configuration. 141

8.8 When a storage server believes that repair is needed, it sends a request to the administrator. After the administrator receives $2f+1$ requests from servers in the current configuration, it creates a new configuration and sends message to servers in the set. The message describes the current state of the log; storage servers fetch the log from members of the previous configuration. 144

9.1 Distributed Directory System Architecture. 150

9.2 The above query states that for a given object identifier select the location-pointers where the remaining number of replicas are less than the low_watermark, thus triggering a repair audit 151

9.3 Directory Data Recovery. a) Using its location-pointers and storage server availability database, the root monitoring server (MIT) knows that there are two replicas remaining. If the low watermark is three, then the root triggers a repair audit. b) The storage servers containing the remaining replicas (Harvard and Texas) cooperate to refresh lost data replicas. 152

9.4 Expanding Radius Heartbeat. Heartbeats initiated by a storage server (e.g. middle server) reach a greater number of additional servers as the heartbeat radius expands. Heartbeats are a form of multicast and reach all servers in the system when the radius is $\log N$. 154

10.1 The path of an (a) `create()`/`put()`/`snapshot()`/`renew()` and (b) `append()`/`truncate()`. 159

10.2 The path of an (a) `repair()` 160

11.1 Aggregation increases system throughput by reducing computation at the data source and in the infrastructure. The base case shows the throughput of a client that stores 4 KB blocks (and a certificate) using `put()` operation, as in a traditional DHT. . . . 166
11.2 The throughput of the system scales with the number of users until resources at the storage servers are saturated. Performing bulk writes using the `put()` interface, the cluster deployment becomes saturated with 48 data sources. Using the `append()` interface, the sustained throughput is much lower because each quorum operation adds only a small amount of data to the log. 167
11.3 Different operations have widely varying latency. The latency is dependent on the amount of data that must be transferred across the network and the amount of communication with the administrator required. The latency CDF of all operations (even the `null()` RPC operation) exhibits a long tail due to load from other, unrelated jobs running on the shared cluster. 168
11.4 Increasing the deployment's tolerance to faults reduces the system throughput since the system must transfer more data with each write operation. 170
11.5 The latency of operations on PlanetLab varies widely depending on the membership and load of a configuration. As an example, this graphs plots the CDF of the latency for appending 32 KB to logs stored in the system. The table highlights key points in the curves. 171
11.6 Quorum Consistency and Availability. (a) Periodic reads show that 94% of quorums were reachable and in a consistent state. Up to 90% of failed checks are due to network errors and timeouts. (b) Server availability trace shows that 97% of quorums were reachable and in a consistent state. This illustrates the increase in performance over (a) where timeouts reduced the percent of measured available quorums. 176
11.7 Number of servers with their Antiquity application available per hour. Additionally, number of failures per hour. Most failures are due to restarting the unresponsive Antiquity instances. As a result, a single server may restart its Antiquity application multiple times per hour if the instance is unresponsive. 177
11.8 Number of replicas created over time due to storing new data and in response to failure. 177

13.1 Design Space for Repair Algorithms. 186

List of Tables

2.1 PlanetLab First Interval Trace Characteristics (All-pairs Ping). Permanent and transient server failure distributions. 29
2.2 PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Permanent and transient server failure distributions. 30
2.3 Storage System Algorithm Parameterization 32
2.4 Existing Storage System Parameterization 32

4.1 Comparison of Replica Placement Strategies. $r_L = 5$ and $n = 11$. 79

5.1 Notation 89

6.1 The certificate present with each operation and stored with each log. It includes fields to bind the log to its owner and other metadata fields. 110
6.2 Operations to `create()`, `append()`, and retrieve data via `get_blocks()` from a secure log. A log is identified by the hash of a public key ($H(PK)$). The `create()` and `append()` operations include a certificate. Further, `append()` requires a verifier of the previous state of the log as a predicate. The `get_blocks()` operation requires two arguments because the system breaks logs into extents and requires both the extent_name and block_name. The `get_map()` retrieves the mappings of a previous extent_counter to previous extent_name. 111

7.1 First-generation distributed hash table (DHT) storage systems use a simple `put()/get()` interface. The `put_hash()` and `put_key()` functions are often combined into a single `put()` function. $H()$ is a secure, one-way hash function; h is a secure hash, as output from $H()$. 119
7.2 To support aggregation of log data, we use an extended API. A log is identified by the hash of a public key ($H(PK)$). Each mutating operation must include a certificate. The `snapshot()` and `truncate()` operations manage the extent chain; the `renew()` operation extends an extent's expiration time. The `get_blocks()` operation requires two arguments because the system implements two-level naming. The `extent_name` is either H(PK) for the log head or verifier for hash-verified extents. . 123

8.1 A configuration defines a set of storage servers that maintain a replicated log. . . . 129

8.2 A soundness proof can be presented by any machine to any other machine in the network to prove that a write was sound. To provide this guarantee, the proof contains a set of q storage server signatures over an append's certificate (Table 6.1) and the storage configuration (Table 8.1). 137

10.1 Breakdown of latencies for all operations. Unless an operation is stated explicitly, `create()` represents all operations that interact with the administrator such as `put()`/`snapshot()`/`renew()`, and `append()` represents all operations that do *not* such as `truncate()`. Total operation latency is $T_{\text{req}} + T_{\text{create\_config}} + T_{\text{quorum}} + T_{\text{resp}}$ for `create()` and $T_{\text{req}} + T_{\text{quorum}} + T_{\text{resp}}$ for `append()`. For all time breakdowns $\mathcal{N}(X)_{a\_b} = (\alpha_{net} + X\beta_{net})$ and $\mathcal{D}(X) = (\alpha_{disk} + X\beta_{disk})$ are the network (from a to b) and disk delays, respectively, where α is the latency, β is the inverse of the bandwidth (bytes per second), and X is the number of bytes. Next, cl =client (or app), gw = gateway, co = coordinator, ad = administrator, and ss = storage server. Finally, $\mathbf{s}$, v, L, and P are the times to sign, verify, DHT `lookup()`, and DHT `publish()`, respectively. Notice that `create()` requires three signatures and `append()` requires two. 161

11.1 Measured breakdown of the median latency times for all operations. For all operations, the client resides in the test cluster and the administrator and storage servers reside in the storage cluster. The average network latency and bandwidth between applications on the test cluster and storage cluster is 1.7 ms and 12.5 MB/s (100 Mbs), respectively. The average latency and bandwidth between applications within the storage cluster is 1.6 ms and 45.0 MB/s (360 Mbs). All data is stored to disk on the storage cluster using BerkeleyDB which has an average latency and bandwidth of 4.1 ms and 17.3 MB/s, respectively. Signature creation/verification takes an average of 6.0/0.6 ms on the test cluster and 3.2/0.6 ms on the storage cluster. Bandwidth of the SHA-1 routine on the storage cluster is 80.0 MB/s. Finally, DHT `lookup()` and DHT `publish()` take an average of 4.2 ms and 7.2 ms, respectively. 169

12.1 System Comparison 180

Acknowledgments

I thank the Lord for blessing me with the opportunity to pursue and finish a doctorate in computer science. Everything I have accomplished has been possible through Him.

John Kubiatowciz, my advisor, has been instrumental in developing my research career. He inspired my research in wide-area fault tolerant storage systems with his OceanStore proposal [K+00]. In that proposal he posed the question: how do we build a system that stores everyone's data, for at least their lifetime, and without losing any information? The question (which seemed obscure at first) is relevant given our increasing dependency on digital data. This thesis is a product of answering that question. More importantly, Kubi has taught me how to ask a research question and how to develop a methodology to answer it.

Anthony Joseph has advised me on research topics and career directions. His endless input and advice was vital to the architecture of OceanStore and Antiquity. He has always been available to refine both oral and written presentations including this thesis. His support has helped define my path.

David Culler has given me excellent advice. He helped develop early OceanStore research ideas while working together at the Berkeley Intel Research Lab. Later, his insights on the cost of storage due to transient failures were vital to this thesis. Furthermore, his letters of recommendation helped with many awards such as the Intel Foundation PhD Fellowship and other opportunities.

John Chuang has taken an active interest in my research and career. He has helped guide me through my qualifying exam, dissertation, and career via letters of recommendation. I am grateful for his commitment.

My office mates, Byung-Gon Chun, Patrick Eaton, Dennis Geels, and Sean Rhea have been great friends. Our relationship in and out of the office has positively affected me. Their influence, feedback, and collaboration can be seen throughout this thesis. Discussions with my office mates along with Emil Ong, (Kelvin) Chiu Wah So, Jeremy Stribling, and Ben Zhao resulted in publications as well.

This thesis is also the product of collaboration with colleagues at other universities. Frank Dabek, Andreas Haeberlen, Emil Sit, Frans Kaashoek, and Robert Morris helped develop many of the fault tolerance and repair techniques presented in Part II. Some of the material resulted in the Carbonite algorithm and was published in NSDI [CDH+06]. Ken Birman and Robbert van Renesse helped clarify some of the ideas about secure logs and dynamic Byzantine quorums in Chapters 7 and 8.

While at Berkeley, many people and organizations have helped shape my career and me as a person. My friend Greg Lawrence has helped me with moral and technical support on a number of occasions. Sheila Humphreys has been my advocate since arriving at Berkeley. Michele de Coteau, Beatriz Lopes-Flores, and Carla Trujillo, have exposed me to many opportunities. Mary Byrnes, Ruth Gjerde, Mary Kelleher-Jones, Peggy Lau, and La Shana Porlaris have helped with many administrative tasks. The Black Graduate Engineering and Science Students (BGESS) has helped keep my life balanced.

As an undergraduate at the University of Washington many people and organizations helped prepare me for graduate school. Debra Friedman, Lori Colliander, and Edward Lazowska saw the potential in me and helped develop my goals. Professors Gaetano Borriello, Carl Ebeling, Hank Levy, and Larry Snyder taught me the excitement of computer science and engineering. Scott Minnix and Lisa Peterson via the Minority in Science and Engineering Program (MSEP) helped prepare me to compete in college. National Society of Black Engineers (NSBE) equipped me with the appropriate tools and insights for my professional career. Finally, the University of Washington football team through Coaches Randy Hart, Jim Lambright, Ron Milus, and Scott Pelluer taught me discipline, integrity, patience, and perseverance.

In High School, my academic drive became apparent during a turning point when I competed for, and won, an internship at the R.S. Dow Neurological Sciences Institute, where I studied with Dr. Neal Barmack. I also learned other vital lessons during this time period. Coach John Eagle taught me how to handle the “sudden changes” in life. Coach Dan Kielty showed me how to be a “student of the game”. Furthermore, friends and their families such as the Leifheit’s, Dicklich’s, and Van Ness’s enriched my life.

Throughout this entire process, Makda Weatherspoon, has been my best friend. She has encouraged, loved, and supported me through the ups and downs of undergraduate and graduate school. She has read and edited this thesis multiple times. Together we have two of the most beautiful children in the world. Makda, Menelik, and Saba are the sun in my sky!

My family has provided the foundation and framework that defines who I am. My parents, Anthony and Sophie Weatherspoon, have been the force behind me. They shaped my character and taught me to be honest, forthright, and conduct myself with the highest integrity. My older brother and sister, Sultan Weatherspoon and Elnora Jimerson, paved the way for me. I hope to provide the same inspiration for my younger brother and sister Ameen Weatherspoon and Megan Rogers. Finally, our family who has been reuniting every year for 105 years has helped me comprehend the value and importance of family.

Portions of this thesis were published in shortened form in the proceedings of USENIX NSDI Conference [CDH+06], USENIX File and Storage Technologies (FAST) Conference [REG+03], IEEE Security in Storage Workshop [EWK05], International Workshop on Peer-To-Peer Systems (IPTPS) [WK02], International Workshop on Future Directions of Distributed Systems (FuDiCo) [WWK02], and International Workshop on Reliable Peer-to-Peer Distributed Systems [WMK02]. My research was supported in part by an Intel Masters Apprenticeship Program (IMAP) scholarship and Intel Foundation PhD Fellowship, as well as summer internships with Intel and IBM.

Part I

Defining Scope of Problem

Chapter 1

Introduction

The preservation of digital data over long periods of time is a challenging endeavor. The amount of such data is increasing as everything from business and legal documents, to medical records, to news and literature, to photos, music and videos are transitioning to digital formats. Systems that store digital assets must ensure durability and integrity of potentially irreplaceable data, and allow users to retrieve data quickly when it is needed.

A variety of approaches for archival storage have been proposed. One recent trend is clear: disks have begun to replace tape as the medium of choice for long-term data preservation [Cor, GSK03, GGL03, GCB+02]. Not only is the cost per bit of storage decreasing faster with disk than with tape, but also the on-line nature of disk-based archival storage leads to greater availability and ease of automatic replication as media fails and hardware changes. However, cheap on-line disk-based storage is not a sufficient solution if disks are colocated in the same machine room, data center, or geographic area. Such a solution cannot tolerate disaster without loss of data. For example, consider a solution that replicates data in physically separated data centers but all located in the same city. If a disaster were to occur (e.g. flood caused by Hurricane Katrina) then data would be lost. Replicating data across the wide-area would help prevent data loss for long-term storage.

Another trend that has appeared in literature but is not universally accepted is durability through geographic-scale replica distribution [CDH+06, DKK+01, HMD05, MMGC02, DR01, REG+03]. By “geographic scale”, we mean the spreading of replicas across multiple states or continents. Advantages of this approach appear to include scalability and resilience to correlated failures such as local disasters. For example, a simulation of the Carbonite algorithm [CDH+06] is able to maintain 100% durability over the course of a year on PlanetLab [BBC+04] despite losing over a third of the servers due to permanent server failure such as disk failure and permanent removal from

the network. Unfortunately, such widely distributed systems suffer from new challenges of security (including malicious components) and automatic management (reliable adaptation to failure in the presence of many individual components).

A geographically spread archival storage system needs to be adaptive and tolerant of the wide-area environment: long latencies, limited access link bandwidth, increased transient failures where data is intact on disk but not immediately available, and malicious agents that attempt to compromise servers and data. In particular, designing such an archival storage system that aggregates disks of a large number of servers spread across the wide-area, for long periods of time, is a challenging pursuit, but a necessary one.

This thesis represents a step towards outlining and addressing the challenges of a distributed wide-area on-line archival storage infrastructure. We assume that such an infrastructure is an essential layer for a variety of applications. We proceed to address two questions: First, how can an archival infrastructure be constructed to provide durability, integrity, and efficiency? Second, what is an appropriate *interface* between applications and an archival infrastructure? Finally, we build such a system called Antiquity to verify the constructions and interface.

1.1 Overview

The task of a distributed wide-area on-line archival storage system is to ensure the durability and integrity of digital data. *Durability* means data stored in the system is not lost due to permanent server failure such as disk failure. *Integrity* means that the state of data stored in the system always reflects changes made by the owner. We discuss the components of durability and integrity further in Sections 1.1.1 and 1.1.2.

1.1.1 Maintaining Durability

In order to maintain durability, the following components need to be addressed: *fault tolerance* and *repair*.

Fault tolerance represents a data object's ability to tolerate permanent server failure without being permanently lost. It is characterized by an object's *configuration*, which defines the type of redundancy (replication or erasure-codes), number of appropriate replicas, and location of those replicas. Both replication and erasure-codes duplicate data in order to reduce the risk of data loss and are considered redundancy. The difference is that replication refers to the process of creating

(a) Fault Tolerance (b) Repair

Figure 1.1: Example Maintaining Durability in a Distributed Wide-area On-line Archival Storage System. (a) fault tolerance defines the system's ability to tolerate server failure without loss of data. It includes choosing the type of redundancy (e.g. replication or erasure-coding), number of replicas, and where to store replicas. (b) Repair is the process of replacing replicas lost to server failure (such as Louisiana). It includes detecting server failure and creating new replicas (such as Georgia).

whole, identical, copies of data. Erasure-coding maps a data object broken into m original fragments (pieces) onto a larger set of n fragments ($n > m$) such that the original fragments can be recovered from a subset of all n fragments. The fraction of the fragments required is called the rate, denoted r[1]. Data loss occurs when all replicas or a sufficient fraction of fragments for erasure-codes are lost due to permanent server failure. In addition to the type of redundancy, the number and location of replicas or fragments are critical to an object's ability to tolerate failure. A fault tolerance algorithm is a set of procedures used to parameterize the components of an object's configuration. Once selected, an object's configuration is static and does not respond to permanent server failure; rather, it tolerates them without data loss.

Repair is the process of replacing replicas lost to permanent server failure. It includes the monitoring of servers and the creation of new replicas when failure occurs. Each time repair is invoked, a *configuration change* occurs since a new set of servers is assigned the responsibility of

[1]Optimal erasure codes such as Reed-Solomon [BKK+95, Pla97, RV97] codes produce $n = m/r$ ($r < 1$) fragments where any m fragments are sufficient to recover the original data object. Unfortunately optimal codes are costly (in terms of memory usage, CPU time or both) when m is large, so near optimal erasure codes such as Tornado codes [LMS+97, LMS+98] are often used. These require $(1+\varepsilon)m$ fragments to recover the data object. Reducing ε can be done at the cost of CPU time. Alternatively, rateless erasure codes such as LT [Lub02], Online [May02], or Raptor [Sho03], codes transform a data object of m fragments into a practically infinite encoded form.

storing replicas. The set of servers in the old and new configuration may overlap or could be completely disjoint. In particular, repair invokes a fault tolerance algorithm to select a new configuration for the object. It then creates and stores replicas on new servers.

Fault tolerance and repair are two different sides of the same coin. Fault tolerance algorithms select a configuration to tolerate failure and are invoked in two situations: when the object is initially inserted into the system and during repair.

Consider the following example to understand the interaction between fault tolerance and repair. Initially, a fault tolerance algorithm selects a configuration that uses a replication redundancy scheme to produce three total replicas for some object. Furthermore, the fault tolerance algorithm selects three servers distributed throughout the wide-area to store replicas. Over time, if a server storing a replica in Louisiana permanently fails, the replica would be lost (Figure 1.1(a)). As a result of the failure, a repair process is triggered to create a new replica, which it then stores on a server in Georgia (Figure 1.1(b)). Repair uses a fault tolerance algorithm to select a new configuration and chooses a new server to host the new replica.

Threats to Durability

There are many threats to durability that complicate the construction of a distributed wide-area on-line archival storage system. The main threat is losing the last copy of an object due to permanent server failure such as disk failure. Bursts of permanent server failure such as those observed on PlanetLab [CDH+06, BBC+04] can leave a data object without any replicas. Efficiently countering this threat to durability involves understanding the parameters of fault tolerance and repair discussed in more depth in Part II.

Another threat to durability is the increase of costs due to *transient server failure* such as server reboot, network and power outage, and software crash. Transient server failure is when a server returns from failure with data intact. For example, PlanetLab experienced 21,255 transient server failures in one year, but only 219 permanent failures. Transient server failure increases costs unnecessarily if the system creates replicas in response to them. Avoiding this cost is difficult because it is not possible to distinguish transient from permanent server failure since they both have the same characteristic. In particular, objects can be durably stored during a transient failure even though the object is not immediately available. For instance, if the only copy of an object is on the disk of a server that is currently powered off, but will someday re-join the system with disk contents intact, then the object is durable but not currently available. As a result, an object is unavailable

during both permanent and transient failure. Since transient server failure does not decrease data durability, creating replicas in response is not necessary. The dilemma is determining when to create replicas without perfect knowledge of which failures are permanent or transient.

1.1.2 Maintaining Integrity

We say that the integrity of data is maintained if the state of data stored in the system always reflects changes made by the owner and cannot be altered by error or malicious agents. We assume each data object is owned by a single principle, which is represented by a public/private key pair. Multiple devices such as a workstation and laptop may possess both keys.

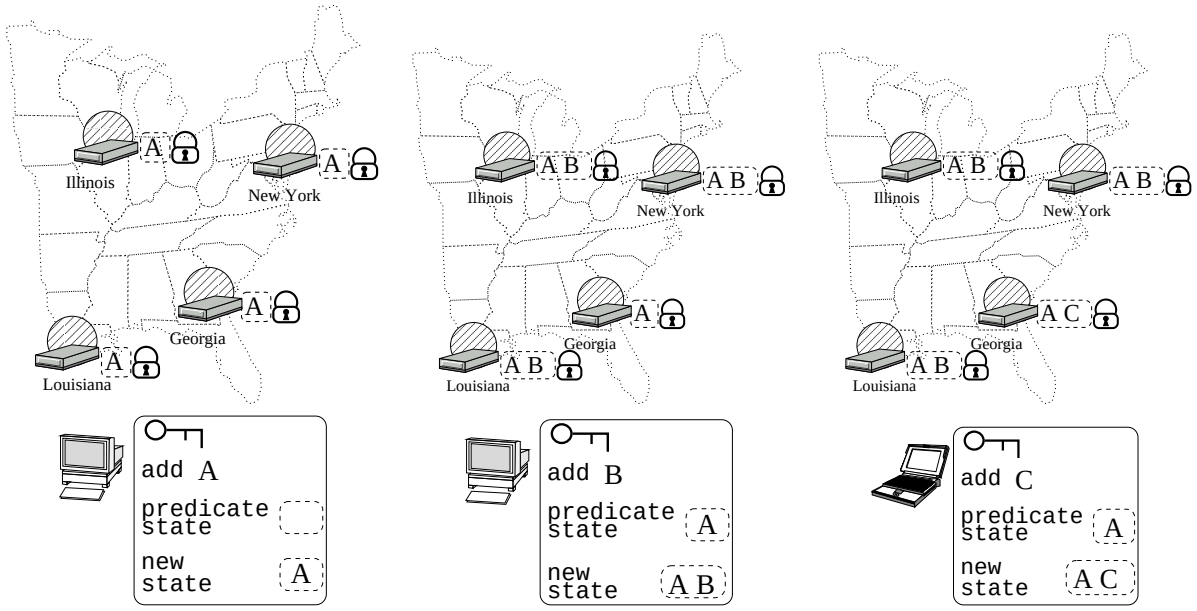
We address three properties of integrity: *non-repudiation*, *data integrity*, and *order integrity*. These properties ensure that only the owner can modify data, returned data is the same as stored, and the last write is returned in subsequent reads, respectively.

Non-repudiably binding data to owner ensures that only an entity possessing the owner's private key can modify data. It includes identifying the owner of each data object and binding owner to the data object and modifications. It ensures servers do not store and cannot present data and changes made by any entity other than the owner. It is necessary since servers that initially store replicas of a data object may not be the same servers that later store the data object and receive modifications.

Data integrity is achieved when returned data is the same as stored data. Mechanics for ensuring data integrity include associating a cryptographically secure hash [NIS94] with each data object. A cryptographically secure hash is a digest (aka checksum, summary, or fingerprint) representation of a data object that makes it difficult for error during network transmission, in storage, or via a malicious attacker to corrupt or alter data without detection.

Order integrity refers to the property by which the last write is returned in subsequent reads. It defines a total order over all modifications to a particular data object. As a result, each server has the ability to accept or reject a modification that cannot be applied. For example, if each modification is assigned a monotonically increasing sequence number, then a server can reject modifications assigned a lower sequence number than the latest modification accepted.

Consider the following usage scenario to understand the properties of integrity. First, assume that an owner's workstation batches updates and periodically (e.g. once an hour) stores data into a distributed wide-area on-line archival storage system. Additionally, the owner occasionally uses a laptop to store modifications directly to the storage system without synchronizing with the



(a) Request Succeeds (b) Request Succeeds (c) Request Fails

Figure 1.2: Example Maintaining Integrity in a Distributed Wide-area On-line Archival Storage System. (a) A data object with the value *A* is replicated onto four servers via a request by a workstation. In (b), the same workstation attempts to add value *B*, predicated on *A* already being stored. The request succeeds since it reaches a threshold of servers (Illinois, Louisiana, and New York). The request did not reach the Georgia server, however, possibly due to network transmission error or transient failure. In (c), a laptop, which possesses the same private key as the workstation, attempts to add value *C*, predicated on *A* already being stored. The request fails since the predicate fails on a threshold of servers. Note that the server in Georgia applies *C* since the predicate matches local state. However, the system should return value *B* in any subsequent reads.

workstation. The owner and data object are non-repudiably bound via a public/private key pair; storage servers identify data objects via a public key and only accept requests signed by the associated private key. As a result, both workstation and laptop are identified as the same owner and are the only entities allowed to modify the data object. With this scenario, the storage system should be capable of maintaining data and order integrity while only accepting modifications signed by the owner despite arbitrary failures such as network error, server failure, or simultaneously submitted and conflicting requests.

Challenges to Integrity from Replication

A distributed wide-area on-line archival storage system replicates data on multiple servers to provide durability. However, maintaining the properties of integrity over replicated data is chal-

lenging. Servers may store inconsistent data object replica state. Cause of inconsistencies could be due to server failure or network transmission error such as dropped, reordered, or delayed messages.

It is even more challenging to maintain order integrity across data object replicas. It requires maintaining consistent state over a threshold of the servers storing a data object replica. As a result, each storage server obeys a consistency protocol and has the ability to accept or reject modifications that cannot be applied to locally stored state. For example, a predicate might be associated with each modification and the predicate must match a secure hash of the currently stored state of the data object before applying modification.

Consider the following illustration to understand the complexity of maintaining integrity over replicated data. In Figure 1.2.(a), a new, empty, data object which only includes a public key is replicated on four storage servers. The owner's workstation uses its private key to sign a request that adds the value *A* to the data object. The request includes a secure hash of the new server state (*A*) to ensure the data integrity. Additionally, the request includes a predicate indicating the previous stored state is empty. The request succeeds and is applied to all four storage servers.

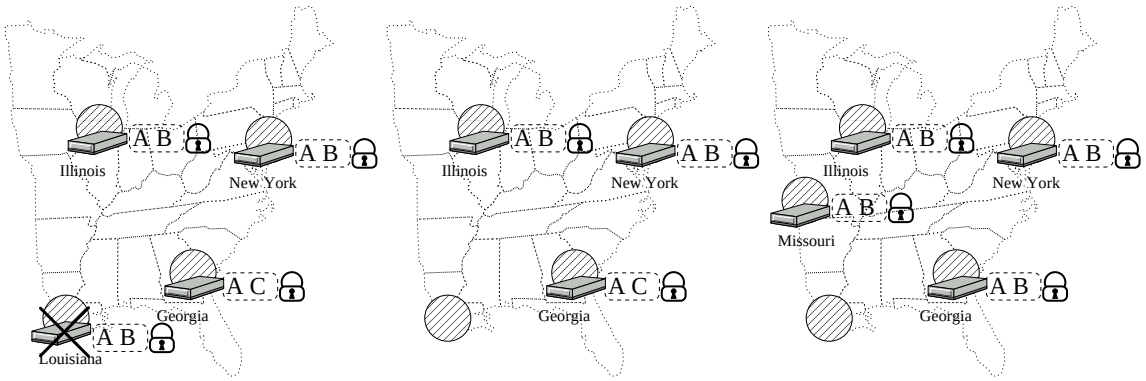
Later, in Figure 1.2.(b), the workstation attempts to add value *B* to the data object, predicated on *A* already being stored. The workstation creates a secure hash of the new server state (*A* and *B*) to ensure the data integrity, cryptographically signs the request to non-repudiably bind it to the owner, and submits it to the storage servers. The request succeeds since it reaches a threshold of servers (Illinois, Louisiana, and New York). The request did not reach the Georgia server, however, possibly due to network transmission error or transient failure.

In Figure 1.2.(c), a different instance of the owner, the laptop, attempts to add value *C*, predicated on *A* already being stored. This request fails on most of the servers, the predicate fails since *A* is not the latest state. As a result, the request fails since a threshold did not apply the request. Note, however, that the server in Georgia applies the request since its state is out-of-date.

Finally, if reads are also performed on a threshold of servers, then the value of the latest write, *B*, will be returned in subsequent reads ensuring order integrity.

Threats to Integrity

The threat to integrity is data corruption on disk or during network transmission and malicious agents that attempt to subvert the system. Moreover, when a system replicates data, it must ensure that replicas are kept consistent and queries are answered in a manner that reflects the true state of the data. Effectively countering these threats to integrity involve many techniques. Cryp-



(a) Permanent Server Failure (b) Permanent Server Failure (c) Repair

Figure 1.3: Example Maintaining Durability and Integrity Together in a Distributed Wide-area Online Archival Storage System. In (a) and (b), a server storing a replica in Louisiana permanently fails. In (c), during repair, the system should initialize the new configuration to the state reflecting the latest successful write, *A B*.

tographic signatures bind data to owner preventing any entity other than the owner from modifying data. Cryptographically secure hashes ensure bits are not corrupted and that data returned is the same as data stored. Finally, consistency protocols such as Byzantine agreement and quorum maintain consistency ensuring that the last write is returned in subsequent reads. The challenge is efficiency since requiring cryptography and consistency in the wide-area can easily be prohibitive.

Alternatively, to make the problems associated with distributed wide-area storage more tractable, many systems eliminate the ability to modify data and store only *immutable* data [CDH+06, DKK+01, HMD05, DR01]. Immutable data is read-only and cannot change. Ensuring data and order integrity is immediate since the data object cannot change. Creating a secure hash one time when the object is first created would be sufficient to ensure that the data returned at some later date is the same as the original data stored. We explore maintaining both immutable data objects that cannot change and *mutable* data objects that can be modified.

1.1.3 Putting It All Together

Combining both durability and integrity poses new challenges to distributed wide-area online archival storage systems. The system must, in aggregate, maintain correct state of data objects even as servers fail, store incorrect local state, or attempt to maliciously alter data. These challenges are compounded by the additional requirements of wide-area, long-term, and efficiency.

In Figure 1.3, we return to the examples originally shown in Figures 1.1 and 1.2 to il-

lustrate the combined requirements of durability and integrity. Recall a fault tolerance algorithm chooses a configuration of four servers to store data object replicas. *B* is the latest successful write to a threshold of servers. Additionally, the server in Georgia stores incorrect state. Later, the server in Louisiana fails as shown in Figure 1.2.(a) and (b). During repair, the system should initialize the new configuration to the latest state, *A B*, shown in Figure 1.2.(c).

In summary, durability via fault tolerance and repair ensures data exists over long periods of time and integrity ensures it correctly reflects changes made only by the owner.

1.2 Challenges

A distributed wide-area on-line archival storage system offers improved durability and increased accessibility but must overcome several disadvantages arising from the distributed environment. We address the challenges associated with durability, fault tolerance and repair, and integrity.

1.2.1 Assumptions

To limit the scope, we make the following assumptions.

First, we assume that eventually all servers permanently fail, that servers are geographically spread across the wide-area, and that data persists longer than the lifetime of any individual server. As a result, the persistence of data is dependent on the system's ability to copy replicas across the wide-area to new servers as old ones fail. We further assume that wide-area access link bandwidth is the critical resource and needs to be efficiently utilized when servers communicate and replicas are created. All durability and integrity guarantees operate under these assumptions.

Second, while we do explore the effects of Byzantine (arbitrary) failures and some correlated failures, we do not explore massively correlated failures, which can result from virus, worm attacks, etc. Such failures can be extremely catastrophic if they cause permanent data loss on a large fraction of disks. We assume, rather, that in aggregate, servers behave correctly and there are a limited number of permanent server failures during some period of time.

Third, we assume storage servers reside in professionally managed sites where sites contribute servers, processor, non-volatile storage, and network bandwidth. We assume server availability, lifetime, storage capacity, and access link bandwidth in professionally managed sites is sufficient to support distributed wide-area on-line archival storage [BR03]. It is the software that allows these

servers and sites to cooperate to maintain data durability and ensure the integrity of data.

Finally, we assume that a particular data object has a single owner but multiple simultaneous writers. For example, an owner might be represented by a public/private key pair and multiple devices such as a workstation and laptop may have access to both keys.

1.2.2 The Fault Tolerance Problem

Fault tolerance is the first key to ensuring data durability. The goal is to tolerate server failure without loss of data. Fault tolerance algorithms must choose the type of redundancy, number of replicas to create, and where to store replicas. We discuss the three components (redundancy, number of replicas, and replica placement) of fault tolerance further.

First, redundancy is the duplication of data in order to reduce the risk of data loss. There are two categories of redundancy: replication and erasure-coding. Replication involves creating whole copies of data. The limitation with replication is that it increases the storage overhead and maintenance bandwidth without comparable increase in fault tolerance. In particular, a linear increase in the replication level results in only a linear increase in the number of failures that can be tolerated. In contrast, erasure-coding involves breaking data into data fragments (pieces of the data object) then creating new redundant fragments that are unique from other data and redundant fragments. Only a fraction of all fragments are required to reconstruct the original data object. All fragments are the same size. Since a fragment may be as large as a whole copy of the data object, erasure-coding is a superset that includes replication. Erasure-codes have a better balance between storage overhead, maintenance bandwidth, and fault tolerance. With a linear increase in storage, the number of server failures tolerated often increases exponentially (e.g. when the number of fragments required to reconstruct the object is greater than one). We demonstrate that erasure-coding is more efficient than replication. However, the choice of redundancy type is a designer decision since replication is simple to use and coding is complex, and the savings are not always worth the increased complexity [RL05, WK02].

Second, fault tolerance is dependent on the number of replicas created. The number of replicas must be configured to cope with a *burst* of failures. It is the size of burst of failures and their probability of occurrence that results in the probability of data loss. If the size of a failure burst exceeds the number of replicas, some objects may be lost. As a result, one could conclude that the highest possible value is desirable. On the other hand, the simultaneous failure of even a large fraction of servers may not destroy any objects, depending on how replicas are placed. Ultimately,

the proper number of replicas to create is related to the burstiness of permanent failures, but other factors such as placement and access link bandwidth limits need to be considered as well.

Finally, replica placement is the process in which servers are selected to store data replicas. The goal of placement is to maximize durability. We show that spreading replicas sets of different objects over many servers increases durability. Less time is required to recover from failure since more servers can assist in repair. The decrease in repair time increases durability since durability is inversely proportional to repair time [PGK88]. Furthermore, we show that a variant of a random replica placement that avoids blacklisted servers and replaces duplicate sites is sufficient to avoid the problems introduced by many observed correlated failures.

Fault Tolerance Insights

The following is a summarized list of insights about fault tolerance. First, with the same storage overhead, erasure-codes tolerate more failures than replication. However, erasure-codes increase the complexity of storage systems, thus designers must weigh efficiency of erasure codes versus the simplicity of replication. Second, increasing the replication level helps cope with bursts of failures. It is the size of burst of failures and probability of occurrence that results in the probability of data loss. Third, less time is required to recover from failure when replica sets for different data objects are spread over many servers so that more servers can assist in repair, thus increase durability. Finally, random replica placement such as one that avoids blacklisted servers and replaces duplicate sites, is sufficient to avoid the problems introduced by the many observed correlated failures.

1.2.3 The Repair Problem

Repair is the other key to ensuring data durability. The goal of repair is to restore the level of fault tolerance by refreshing lost redundancy before data is lost due to permanent failures. We assume replicas are geographically spread, thus the cost of creating new replicas requires wide-area bandwidth, a critical resource. Monitored information, which measures the number of available replicas, is the basis for initiating repair. However, this monitored information is imprecise since replicas can be durably stored (e.g. exist on a server's disk) but not immediately available (e.g. server powered off), hence transient failure. Transient failure does not decrease the number of replicas that are durably stored, therefore it is not necessary to create replicas in response to transient failure to maintain a target level of durability. Furthermore, users of many Internet applications can

tolerate some unavailability. For example, requested data is readable eventually, as long as it is stored durably.

Ideally, replicas would be created in response *only* to permanent server failure. A hypothetical system that can differentiate permanent from transient failures using an oracle could react only to permanent failures. However, it is not possible to distinguish the two failure types using only remote network measurements (e.g. `ping`). Initiating repair after failure, whether permanent or transient, is the method currently used by most existing systems since they are limited to network measurements. This method may serve as a solution but proves to be costly.

Transient failures are common in wide-area systems (e.g. 21,255 transient versus 219 permanent failures in one year on PlanetLab [CDH+06]). Many replication algorithms waste bandwidth by making unneeded replicas. For example, the initial replication algorithm used by many distributed hash table (DHT) storage systems such as DHash [Cat03], OpenDHT [RGK+05], and PAST [DR01] turned out to be costly [CDH+06, WCSK05]. The problem was that their designs were driven by the goal of achieving 100% availability; this decision caused them to waste bandwidth by creating new replicas in response to temporary failures. Their designs and similar ones (such as Total Recall [BTC+04]) are more than what is required for durability.

Since we assume that wide-area bandwidth is a critical resource, a system should attempt to minimize repair costs due to failure while maintaining a target level of durability. A key technique to reduce repair costs is to reduce the number of replicas created in response to transient failures. One solution requires an increase in the time to detect a failure. Such a solution does not respond to many transient failures since a server that transiently fails might return before a failure is detected. However, durability is decreased since the "window of vulnerability" is increased. The larger failure detection time subjects data to loss from additional permanent server failure. Alternatively, instead of increasing the failure detection time, another solution increases the number of replicas. The solution supplements the replication level with extra replicas that are *not* required for durability, but instead are required to be simultaneously unavailable before repair is initiated. For example, assume the replication level required for a target durability is five replicas, then this solution might add two more replicas for a total of seven replicas. Repair is invoked when four replicas (or less) are available , thus this solution would be invoked when *three* (or more) replicas are simultaneously unavailable. In general, as the number of replicas required to be simultaneously unavailable increases, the probability that they are all unavailable due to transient failure decreases exponentially. As a result, increased replication can be used to decrease bandwidth usage due to transient failures.

The solution of using increased replication to reduce repair costs depends on whether data is mutable (data can change) or immutable, (data is read-only and cannot change). For mutable data, servers that return from failure need to either be updated or removed from the replica set if a write occurred while the server was unavailable. To reduce repair cost, the system must estimate the number of replicas that are required to be simultaneously unavailable in order to initiate repair. This estimate is based on system measurements such as average server availability, lifetime, amount of data, and bandwidth to support target durability level. For immutable data, however, reintegrating replicas from transient failures into replica sets minimizes the number of copies created incorrectly due to transient failures. The result is that the system performs some extra work for each object early in its life creating replicas in response to transient failures, but over the long term creates new copies of the object only as fast as it suffers permanent failures. Replicas created in response to transient failure and reintegrated into the replica set insulate the data object from future transient failures.

Repair Insights

The insights of repair can be summarized as follows. First, durability is a more practical and useful goal than availability for applications that store objects (as opposed to caching objects) for long periods of time. Since data is not lost during transient failures, which are common in the wide-area, the cost (such as the number of replicas created per unit time) can be reduced via maintaining high durability versus high availability. Second, the main goal of a durability algorithm should be to create new copies of an object faster than they are destroyed by permanent server failures. The choice of how replicas are distributed among servers can make this task easier. Finally, extra replicas beyond what is required for durability reduce the cost due to transient failure (e.g. number of replicas created). For mutable data, estimating the number of extra replicas minimizes unnecessary copying. For immutable data, reintegrating returning replicas is key to avoiding unnecessary copying.

1.2.4 The Integrity Problem

In addition to the durability requirements of fault tolerance and repair, storage systems must maintain the integrity of data. The problem is we assume any server may behave in arbitrary, Byzantine, ways. A server may be in an arbitrarily undefined or malicious state due to a network error, storage corruption, software bug, or compromise. As a result, a server may modify data in

ways not authorized by the owner of the data, corrupt data on disk or during transmission, or by not following protocol. For example, nearly a third of the PlanetLab servers were compromised in December of 2003 when an attacker exploited a kernel vulnerability [WCSK05].

Since we assume that any fraction of the servers may exhibit Byzantine behavior, it is difficult to develop mechanisms and protocols that ensure the integrity of data in this environment. Specifically, ensuring the integrity of data includes ensuring three properties: non-repudiability, data integrity, and order integrity. We discuss these three properties and viable solutions that ensure them further below.

First, the non-repudiability property ensures that only the owner (or agent of the owner) can modify mutable data. The approach most commonly proposed for assuring this property is to include a cryptographically signed certificate with each data object. The certificate provides a secure, non-repudiable binding between the data and its owner. The certificate remains colocated with the data, even as the data is replicated or transferred.

While this solution is conceptually simple, an efficient implementation has proved elusive. One impediment is the time it takes the client to sign all of the certificates. In fact, some designers have rejected the solution by reasoning that the cost of producing certificates is prohibitively expensive [FKM00]. To illustrate this problem, assume an application running on a 3 GHz processor wishes to store 1 TB of data. If the data is divided into 8 KB blocks and certificates are created using 1024-bit RSA cryptography and a single processor, it would take more than *six days* to create certificates for the data [2]. A hardware accelerator solution can reduce this signature creation time for an increase in financial cost; for example, using six processor cores in parallel with cryptographic co-processors per core can reduce the time by a factor of 32 from six days to six hours.

Instead of designing more expensive hardware solutions, however, we present a solution that addresses the efficiency challenges described above by exploiting aggregation. For instance, consider an application storing 1 TB of data into a system that aggregates data into 4 MB containers. A client machine with a 3 GHz processor could create the certificates in *17 minutes*, a reduction of three orders of magnitude over a system that implements the certificate-per-block approach. Further, using the hardware accelerator solution described above would reduce the time to create certificates to one-half of one minute.

Aggregation by itself, however, is not a sufficient solution since clients should be able to add data to the system without local buffering. The design, then, should aggregate small blocks of

[2] A single 3 GHz Pentium-class processor can create a signature in 4 ms, as measured with the command `openssl speed rsa1024`.

data into larger containers (to amortize the cost of creating and managing certificates) while simultaneously supporting incremental updates (to obviate data buffering at clients) and fine-granularity access (to allow clients to retrieve exactly the data they need). Moreover, the design should allow any member of the storage system to identify, in a secure and non-repudiable fashion, the owner of each piece of data stored in the system.

Second, a system ensures the data integrity property as long as returned data is the same as stored. It is a straight forward application of a secure hash function to ensure data integrity of strictly replicated data. Erasure-codes, however, are more difficult. To apply a secure hash when erasure-coding is used, many systems either require data to be reconstructed from erasure-coded fragments (e.g. DHash [Cat03]) or associate a cryptographically signed certificate that includes a secure hash of all fragments (e.g. Glacier [HMD05]). The problem with the former solution is that a storage server cannot locally verify the integrity of a fragment. The problem with the latter solution is that more bandwidth and storage may be required to store the certificate since the certificate may be larger than a single fragment and each server must store the certificate along with the fragment. Additionally, significantly more processor time is required to create a signature in addition to a hash. We present an algorithm where each erasure-coded fragment, as well as the object itself, can be self-verified by any component in the system using a single secure hash.

Finally, consistency protocols such as Byzantine agreement and quorums ensure the order integrity property that the last write is returned in subsequent reads. Maintaining order integrity is challenging because a threshold of an object's replica servers need to always be available and agree on a value in the midst of arbitrary failure. Once a threshold is no longer available, other servers need to be recruited and integrated into the replica set, *configuration change*. It is this configuration change process while maintaining consistency that is difficult. Previous solutions either do not allow configurations to change or do not guarantee that successful writes are maintained as configurations do change over time. We present a configuration change protocol that maintains order integrity when configurations change even if a threshold is unavailable.

Solving the Integrity Problem with a Secure Log

Basing the design of a distributed wide-area on-line archival storage system on a secure log can solve the integrity problem while incorporating the insights of fault tolerance and repair. Our basic premise is that a secure log provides an ideal primitive for implementing an archival storage infrastructure. A log's structure is simple and its security properties can be verified [LKMS04,

MRC+97, Mer88, MMGC02, SK98]. Only a single interface, `append()`, is provided to modify the log, and all mutations occur at a single point—the log head. A system can secure the log head by requiring that all `append()` operations be signed by the private key of the log owner. If each log element is named individually, random access `get()` provides quick data retrieval. Although simple, this interface is powerful enough to implement a variety of interesting applications.

In Part III, we show how to construct a Byzantine-fault-tolerant, efficient, and wide-area archival system from a secure log. Such an archival system is intended to be a component of a larger application. While a secure log is conceptually simple, replicating the log in a distributed storage system has proved challenging [MMGC02, REG+03]. We describe a system design that combines this log interface with three technologies: quorums, quorum repair, and aggregation. Dynamic Byzantine fault-tolerant quorums ensure consistency of the log heads. Data integrity is assured at both the block and container granularity. We provide data durability with an algorithm that repairs quorums when replicas fail. Finally, aggregation reduces communication costs while maintaining fine-granularity access for clients.

The design of an archival storage system that exploits a secure log has the following contributions. First, a secure-log based archival storage system maintains the integrity of data. Second, a consistency protocol based on dynamic Byzantine fault-tolerant quorums that works well in the wide-area. Third, a Dynamic Byzantine quorum repair protocol that responds to failure and continuously maintains replication and consistency. Finally, an operational prototype that combines these features and is currently running in the wide area.

1.3 Architecture for a Solution: Antiquity Prototype

The conceptual insights and solutions described in Section 1.2 have been embodied in an architecture that we have developed. The first implementation of this architecture is called Antiquity. Antiquity supports a secure log abstraction, only the owner of the log can append new data blocks to the head of the log. It efficiently stores the log as servers fail and ensures the data and order integrity overtime.

We demonstrate that a narrow interface simplifies the design of a storage system. In particular, Antiquity's design and implementation combines the log interface with three technologies—dynamic Byzantine fault-tolerant quorums, quorum repair, and aggregation—to store replicated logs, to enforce consistent append, to provide random-access reads, to ensure durability, and to store and update the log efficiently.

Experience with a deployment of the prototype shows that Antiquity's design is robust. It has been running in the wide-area for over two months on 400+ PlanetLab servers maintaining nearly 20,000 logs containing more than 84 GB of unique data. 94% of the logs are in a consistent state. 100% of logs are durable, though 6% do not have a quorum of servers immediately available due to transient failures. The prototype maintains a high degree of consistency and availability due to the quorum repair protocol despite the constant churn of servers (a quarter of the servers experience a failure every hour).

1.4 Historical Perspective

This thesis is a product of the OceanStore proposal [K+00]. OceanStore is an Internet-scale distributed data store designed to securely provide continuous access to persistent information. Unlike many previous distributed storage systems, OceanStore constructs a reliable and secure storage infrastructure from many untrusted servers. Data is protected via redundancy and cryptographic techniques. Although many servers may be corrupted or compromised at a given time, the aggregate behavior of the complete system provides a stable storage substrate to users. The challenge for OceanStore, then, is to design a system that provides an expressive storage interface to users while guaranteeing high durability atop an untrusted and constantly changing base.

In addition to providing the motivation for this thesis, OceanStore provides the design and implementation experience necessary to investigate the problems posed by this thesis. In particular, Pond [REG+03] is the OceanStore prototype that precedes Antiquity. It contains many of the features of a complete OceanStore system including location-independent routing via Tapestry [ZHS+04, ZJK01] and Bamboo [RGRK04], Byzantine fault-tolerant update serialization and commitment, push-based update of cached copies through an overlay multicast network, and continuous archiving to erasure-coded form. Every Pond server implements each of these subsystems as a stage, a self-contained component with its own state and thread pool, which is a good mechanism to modularize and integrate the subsystems together. Stages communicate with each other by sending events. Most importantly, Pond contains sufficient implementation and integration of the OceanStore design to give a reasonable estimate of the performance of a full system. For instance, in the wide-area, Pond outperforms NFS on the read-intensive phase of the Andrew benchmark, but underperforms NFS on the write-intensive phase. Microbenchmarks show that write performance is limited by the speed of erasure coding and threshold signature generation.

Though experience implementing Pond (and its predecessor Puddle) is necessary for this

thesis, maintaining an integrated solution where each subsystem needs to be tolerant of attack and restartable due to server failure is difficult. Instead of a fully integrated solution, breaking OceanStore into layers may be more attainable. In fact, the secure log interface implemented by Antiquity is a result of breaking OceanStore into layers. In particular, a component of OceanStore is the primary replica implemented as a Byzantine fault-tolerant agreement process. This primary replica serializes and cryptographically signs all updates. Given this total order of all updates, the question is how to durably store and maintain the order? Furthermore, what should be the interface to this underlying storage system? An append-only secure log answered both questions. The secure log structure assists the storage system in durably maintaining the order over time. The append-only interface allows a client (such as OceanStore's primary replica) to consistently add more data to the storage system over time. Finally, when data is read from the storage system at a later time, the interface and protocols ensure that data will be returned and that returned data is the same as stored.

Another aspect of OceanStore from which this thesis borrows is the notion of a responsible party. A responsible party is financially responsible for the integrity of data and selects sets of servers to obey protocol and host data replicas. Superficially, the responsible party seems to introduce a single point of failure into the OceanStore design. While this is true to an extent, it is a limited one. First, there can be more than one responsible party in the system; the role of the responsible party thus can scale well. Second, the responsible party's state can be stored in the system. Thus, the durability of the state can be assured like any other data object. If the responsible party fails, a new one could be created using the state stored in the system. Third, the state of the responsible party can be cached to reduce the query load on it. Finally, the responsible party can be implemented as a replicated service to improve availability further. Antiquity implements the responsible party as an *administrator*, which is consulted to create a secure log and select a set of servers to store log replicas.

In summary, this thesis owes credit to the OceanStore project for providing the initial motivation and experience necessary to investigate, design, and construct distributed wide-area online archival storage systems.

1.5 Lessons Learned

In this section we discuss lessons learned from our experience investigating this thesis.

First, mechanisms for *durability* should be separated from mechanisms for *latency reduction*. For instance, erasure-resilient coding should be utilized for durability, while replicas (i.e.

caching) should be utilized for latency reduction. The advantage of this organization is that replicas utilized for caching are *soft-state* and can be constructed and destroyed as necessary to meet the needs of temporal locality. Further, prefetching can be used to reconstruct replicas from fragments in advance of their use. Such a hybrid architecture is illustrated in Figure 4.3. This is similar to what is provided by OceanStore [K+00, REG+03].

Second, a random placement policy—that avoids blacklisted servers and duplicate sites—-is sufficient to avoid many observed correlated failures. The reason random is effective is that a significant fraction of correlated failures involve a small number of servers and are often predictable (e.g. within same site) [NYGS06]. Furthermore, large correlated events that cause many servers to fail simultaneously occur very infrequently and are unpredictable [NYGS06]; as a result, it is often not possible to avoid these large correlated events. Since most correlated failures are small (involve few servers), however, they are not likely to destroy all data replicas for a particular object and often can be avoided with simple policies. In Section 4.3.2, we demonstrate that a random placement policy (with small optimizations) has similar performance to a clairvoyant placement that knows the future time that servers fail and can avoid correlated failure.

Third, Byzantine fault-tolerant agreement- and quorum-based protocols ensure consistency of replicated state; however, quorum-based protocols are easier to implement than agreement. The difference between the two is that Byzantine fault-tolerant agreement-based protocols use communication between replicas to agree on a proposed ordering of requests; whereas, in Byzantine fault-tolerant quorum-based protocols, clients choose the order and contact replicas directly to optimistically execute operations [CML+06]. It is the selection of an ordering in the mist of failure and attack that make agreement-based protocols difficult to implement. For instance, neither Pond [REG+03] or Castro and Liskov [CL99] initially implemented view changes, which is required to tolerate failure. Quorum-based protocols on the other hand do not require replicas to order requests, instead clients provide the order. Reducing complexity of implementation is the reason Antiquity implements a quorum-based protocol instead of an agreement-based protocol.

Fourth, the use of an administrator significantly reduces the complexity of the overall design of a distributed wide-area on-line archival storage system and reflects the design of other storage systems (such as cluster based storage designs). The storage system can be audited for correctness and ensure that integrity of data is maintained since the administrator authorizes clients to utilize storage resources and selects servers to obey protocol and maintain replicated state. Without an administrator, designs such as Antiquity's could become significantly more complex and difficult to implement.

Finally, storage systems should decouple the infrastructure's unit of management (e.g. extent) from the client's unit of access (e.g. data block). As a result, the storage infrastructure can amortize management costs over larger collections of data while clients can access smaller blocks of data. For instance, Pond maintains (replicated) location-pointers to track the availability of every fragment of every block. However, resource (computation, storage and bandwidth) consumption maintaining location-pointers dominates resource consumption used to maintain actual data. As a result, Antiquity aggregates blocks into containers called extents and maintains metadata on an extent basis reducing resource consumption significantly. Further, the design supports incremental updates (to obviate data buffering at clients) and fine-granularity access (to allow clients to retrieve exactly the data they need).

1.6 Contributions

We make several contributions in this work.

First, **we explore the parameterization space of fault tolerance algorithms and associated durability**. We show that erasure-coding reduces the bandwidth costs to maintain a target level of durability when compared to replication. Alternatively, for the same storage overhead and bandwidth costs, erasure-coding can maintain a significantly higher level of durability than replication. Further, we show durability is related to the distribution of failure bursts. Next, we show that random placement is sufficient to increase durability via reduced repair time and avoid many correlated failures. Additionally, we present a unified view of existing wide-area storage systems and evaluate the long-term maintenance costs of the systems using a trace-driven simulation.

Second, **we show how to reduce costs due to transient failures.** We demonstrate a principled way to estimate the amount of extra replication required to reduce repair costs due to transient failures. Further, when data is immutable, we show that the system can limit the number of unnecessary copies made due to transient failures by ensuring that recovered copies are integrated in place into the replica set. The result is that the system performs some extra work for each object early in its life, but over the long term creates new copies of the object only as fast as it suffers permanent failures.

Third, **we show how a secure log can solve the data integrity problem.** We demonstrate how the narrow interface of a secure, append-only log simplifies the design of wide-area distributed storage systems. The system inherits the security and integrity properties of the log. We describe how to replicate the log for increased durability while ensuring consistency among the replicas. We

present a repair algorithm that maintains sufficient replication levels as machines fail. Finally, the design uses aggregation to improve efficiency.

Finally, **we describe the design and evaluation of Antiquity Prototype.** Antiquity is an implementation of a distributed on-line archival storage system that exploits a secure log interface. It efficiently maintains the mutable log head and all other immutable components applying insights and design points from exploring the fault tolerance, repair, and integrity problems.

1.7 Summary

As the amount of digital assets increase, systems that ensure the durability, integrity, and accessibility of digital data become increasingly important. Distributed on-line archival storage systems are designed for this very purpose. In this thesis, we explore several important problems that must be addressed to build such systems. We start in Part II where we explore how to efficiently maintain durability via fault tolerance and repair. Next, in Part III, we describe an architecture that exploits a secure log to solve the integrity problem. We then apply the insights and architecture to a Prototype called Antiquity in Part IV. Antiquity efficiently maintains the durability and integrity of data. Finally, in Part V, we discuss related and future work and conclude.

Chapter 2

Methodology

The software and algorithms of a distributed on-line archival storage system allow servers to cooperate in order to maintain the durability and integrity of data. The behavior of the software and algorithms, however, depends on the environment in which they are used. An environment is a set of circumstances and conditions under which a server operates. For example, environments with high disk failure rates or low network access link speeds make it difficult for any system to maintain durability [BR03].

To gain a deeper understanding and intuition about algorithmic design decisions and associated costs, we use traces of both existing and synthetic wide-area environment characteristics. These characteristics vary by the rate and distribution of permanent and transient server failures. Additionally, they vary by access link bandwidth. The target environment characteristics are that of servers residing in managed sites such as universities, companies, etc.

To measure and compare algorithms, we use environment characteristics to drive a series of simulations. We use traces of permanent and transient server failure to drive an event-based simulator. A server is affected by three events in a trace: `join`, `fail`, `crash`. In the simulator, a server is added and all content stored by the server is available at the time of a `join` in the trace. A server is removed and all content stored by the server is unavailable at the time of a `fail` or `crash`. Furthermore, `crash` permanently removes all content stored by the server. Servers that are not available in the trace are not available in the simulator (and visa versa). Figure 2.1 illustrates a server failure traces. Finally, in the simulator, each server has unlimited disk capacity, but limited link bandwidth.

A distributed wide-area storage algorithm maintains data durability as servers fail. Each algorithm is represented as a set of parameters. Parameters include redundancy type, target repli-

```
1109635207  join   219.243.200.37
1109635207  join   132.239.17.226
1111558805  fail   219.243.200.37
1111559207  join   219.243.200.37
1112813519  crash  132.239.17.226
```

Figure 2.1: Example Server Failure Trace.

cation level, replica placement strategy, failure detection time, number of extra replicas beyond replication level, and whether to reintegrate replicas or not. The simulator measures the cost that storage algorithms incur to maintain data (e.g. cumulative number of replicas created over length of trace). Further, it measures the number of objects permanently lost. Together, cost and durability metrics describe the effectiveness of a particular algorithm in maintaining data over a particular server failure trace.

In general, simulation provides a way to study system algorithmic design alternatives in a controlled environment. Simulation facilitates exploring system configurations that are difficult to physically construct. Simulation can observe interactions that are difficult to capture in a live system. Further, simulation can compare cost tradeoffs over time. For example, Total Recall [BTC+04] showed via simulation that lazy repair could mask transient failures by delaying triggering repair, which reduced the cost of maintaining durability. Simulations drive our analysis and comparisons in Part II.

After simulating system behavior, we use the insights gained from simulation to design, implement, and deploy best approaches. We use both emulated and real environments to measure performance of a deployed system in Part IV. We use a real deployment running and storing data on PlanetLab [BBC+04] to evaluate the efficacy of proposed algorithms. Furthermore, we measure the performance of the deployed system in alternative environments where we emulate different server failure patterns such as the failure trace used in simulation.

The failure characteristics are described further in Section 2.1. In Section 2.2, we describe the simulation, emulation, and deployment environments.

2.1 Failure Characteristics

We use permanent and transient server failure characteristics from both real and synthetic sources. First, we use PlanetLab [BBC+04] to create failure traces of an existing wide-area environment. PlanetLab is a large ($>$ 600 server) research testbed with servers located in many universities

and companies from around the world. It is a distributed collection of servers that have been monitored for long periods of time. We use this monitored data to construct a realistic trace of failures in a mostly managed environment.

Furthermore, traces of PlanetLab server failures can be divided into two distinct intervals. The first interval contains many correlated failures due to software and disk upgrades, as well as compromise of the system. The second interval contains less correlated failure since system maintenance and operation had matured. We expect the second interval to be more typical of a storage environment.

Finally, for explanatory purposes, we also use a synthetic trace that makes some of the underlying trends more visible. For example, we may increase the length of the trace, increase the failure density, or remove transient failures, etc.

The rest of this section is organized as follows. First we discuss the failure collection technology in Section 2.1.1. Then we discuss the three failure trace characteristics in Sections 2.1.2, 2.1.3, and 2.1.4.

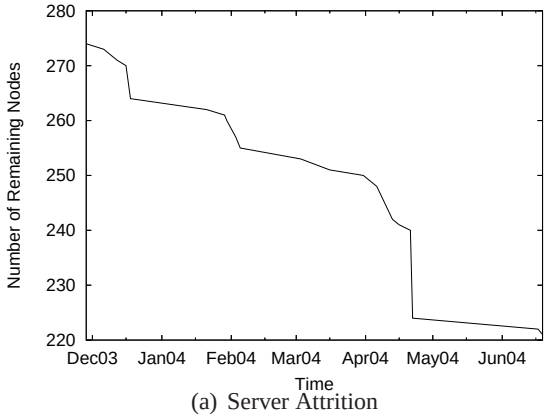
2.1.1 Analyzing the Behavior of PlanetLab

We use two different data sets to characterize the behavior of PlanetLab: all-pairs ping [Str] and CoMon project [PP06] plus PlanetLab Central [PCAR02] (CoMon+PLC) data sets. The all-pairs ping data set characterizes the first interval of PlanetLab operations. The CoMon+PLC data set characterizes the second interval. The difference between the two data sets is how and when the data was collected. We explain further below.

All-Pairs Ping Data Set

The all-pairs ping data set provides failure data for the first interval of PlanetLab operations. It spans from February 16, 2003 to June 25, 2005 and includes a total of 694 servers in that time period. It collects minimum, average, and maximum ping times (over 10 attempts) between all pairs of servers in PlanetLab. Measurements were taken and collected approximately every 15 minutes from each server. The 15 minute ping period does not contain enough fidelity to detect transient failures less than 15 minutes. Measurements were taken locally from individual servers' perspective, stored locally, and periodically archived at a central location. Failed ping attempts were also recorded.

To create a trace from the all-pairs ping data set, we need to determine the times during



(a) Server Attrition

UpperBound	LowerBound
951 days	663 days

(b) Expected Lifetime Estimates

Figure 2.2: Server Attrition of All-pairs Ping Data Set.

which servers transiently and permanently fail. The server's transition from available to unavailable defines a failure, which might be transient if the server returns later. We use at least a single successful ping from at least one other server to determine that a server is available. On the other hand, if there are no successful pings from any server, then the server is unavailable. This single ping and server method of determining server availability was used by Chun and Vahdat [CV03]; a single non-faulty path is sufficient for many routing algorithms to allow servers to communicate. For example, algorithms exist for servers to communicate in networks with non-transitive connectivity [And04, FLRS05].

All-pairs ping does not measure permanent failure. Instead, it measures only the availability of a server name (i.e. availability of an IP address) and not the existence of data on a server. As a result, all-pairs ping was used to produce an estimated upper bound on server lifetimes. We used a technique described by Bolosky et al. [BDET00] to estimate the expected server lifetime based on server attrition. In particular, if servers have deterministic lifetimes, then the rate of attrition is constant, and the count of remaining servers decays linearly. The expected server lifetime (meaning the lifetime of the servers IP address, not physical hardware) is the time until this count reaches zero [BDET00]. We counted the number of remaining servers that started before December 5, 2003 and permanently failed before July 1, 2004. The expected server lifetime of a PlanetLab server is 951 days (Table 2.2(b)). Figure 2.2(a) shows the server attrition. Furthermore, we computed an estimated lower bound of a server lifetime by supplementing the trace with a disk failure distribu-

tion obtained from [PH02] (Table 2.2(b)). In our experiments, the expected lifetime of a server lies between the upper and lower bound.

Another limitation of the all-pairs ping data set is no data exists between December 17, 2003 and January 20, 2004 due to a near simultaneous compromise and upgrade of PlanetLab. In particular, Figure 2.3(a) shows that 150 servers existed on December 17, 2003 and 200 existed on January 20, 2004, but no ping data was collected in between the dates above.

CoMon+PLC data set

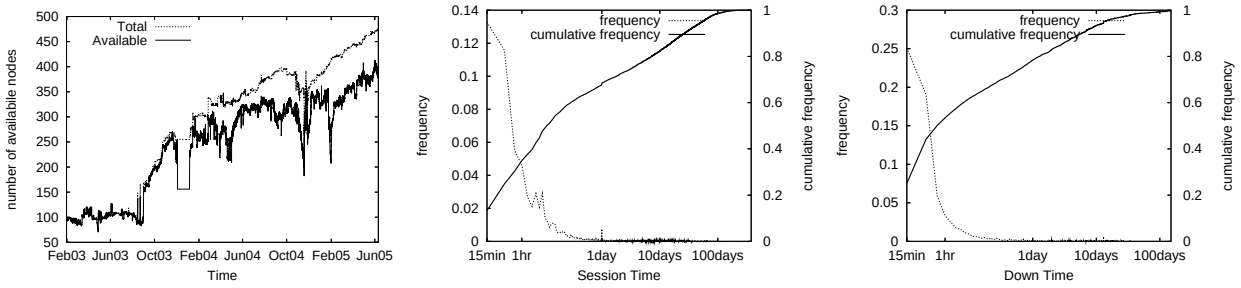
The CoMon+PLC data set provides failure data for the second interval of PlanetLab operations. It is from March 1, 2005 to February 28, 2006 and includes a total of 632 servers. We use historical data collected by the CoMon project to identify transient failures. CoMon has archival records collected on average every 5 minutes that includes the uptime as reported by the system uptime counter on each server. We use resets of this counter to detect reboots, and we estimate the time when the server became unreachable based on the last time CoMon was able to successfully contact the server. This allows us to pinpoint failures without depending on the reachability of the server from the CoMon monitoring site.

We define a disk failure to be any permanent loss of disk contents, due to disk hardware failure or because its contents are erased accidentally or intentionally. In order to identify disk failures, the CoMon measurements were supplemented with event logs from PlanetLab Central. This database automatically records each time a PlanetLab server is reinstalled (e.g., for an upgrade, or after a disk is replaced following a failure). The machine is then considered offline until it is assigned a regular boot state in the database.

2.1.2 PlanetLab First Interval: Insights into Correlated Failures

Figure 2.3 shows the PlanetLab first interval trace characteristics using the all-pairs ping data set. Figure 2.3(a) shows the (total and available) number of servers versus time. It demonstrates system growth over time. More importantly, it pictorially shows the number of servers that we simulated at each time instance in our trace.

Figures 2.3(b) and (c) show the frequency and cumulative frequency for the sessiontime and downtime, respectively. Note that the frequency uses the left y-axis and cumulative frequency uses the right. A *sessiontime* is one contiguous interval of time when a server is available. In contrast, a *downtime* is one contiguous interval of time when a server is unavailable. Sessiontime



(a) Number of available servers v Time (b) Sessiontime CDF and PDF (c) Downtime CDF and PDF

Distribution	Mean(d)	Stddev(d)	Median(m)	Mode(m)	Min(m)	90th(d)	99th(d)	Max(d)
Session	8.5	23.5	180	15	15	26.6	118.0	376.7
Down	3.5	16.9	45	15	15	5.9	73.2	388.7
Availability	0.700	0.249	0.757	1.000	0.002	0.977	1.000	1.000

(d) Sessiontime, Downtime, and Availability Statistics (d=days, m=mins)

Figure 2.3: PlanetLab first Interval Characteristics (All-pairs Ping). Sessiontime, downtimes, and availability distributions.

and downtime are commonly referred in the storage literature as a time-to-failure (TTF) and time-to-repair (TTR), respectively. Additionally, the average sessiontime and downtime is the mean-TTF and mean-TTR (MTTF and MTTR), respectively. A server's *lifetime* is comprised of a number of interchanging sessiontimes and downtimes. The mean sessiontime was 204.4 hours (8.5 days) and mean downtime was 82.8 hours (3.5 days). Both the sessiontime and downtime distributions were long tailed and the median times were 3 hours and 0.75 hours, respectively.

The sessiontimes decreased dramatically between October 2004 and March 2005 due to multiple kernel bugs that caused chronic reboot of PlanetLab servers (shown in Figure 2.3(a)). The chronic reboots within the last six months of the trace doubled the total number of sessions with mostly short sessiontimes. In particular, the median sessiontime decreased from 55.8 hours (2.3 days) between February 2003 and October 2004 to 3 hours between February 2003 and June 2005. Despite the decrease in sessiontimes, we continue to use PlanetLab as an example wide-area system. By doing so, we show how storage systems should adapt to changes over time without loss of data or increase in communication costs.

Figure 2.3(d) summarizes the sessiontime, downtime, and server availability statistics. Availability is dependent on the sessiontime and downtime. It is the percent of time that a server is up (i.e. total sum of the sessiontime divided by the lifetime or the average sessiontime divided by the sum of the average sessiontime and downtime) and is equivalent to the more commonly known expression $\frac{MTTF}{MTTF+MTTR}$. We measured that 50% of the servers have an availability of 75.7% or

Dates	16 February 2003 – 25 June 2005
Number of hosts	694
Number of transient failures	23903
Number of disk failures	308
Transient host downtime (s)	2700, 297994, 507600
Any failure interarrival (s)	868, 3079, 6300
Permanent failures interarrival (s) (Median/Mean/90th percentile)	11702, 234965, 629045

Table 2.1: PlanetLab First Interval Trace Characteristics (All-pairs Ping). Permanent and transient server failure distributions.

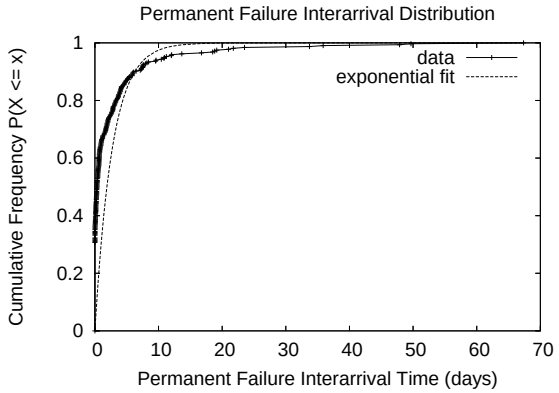


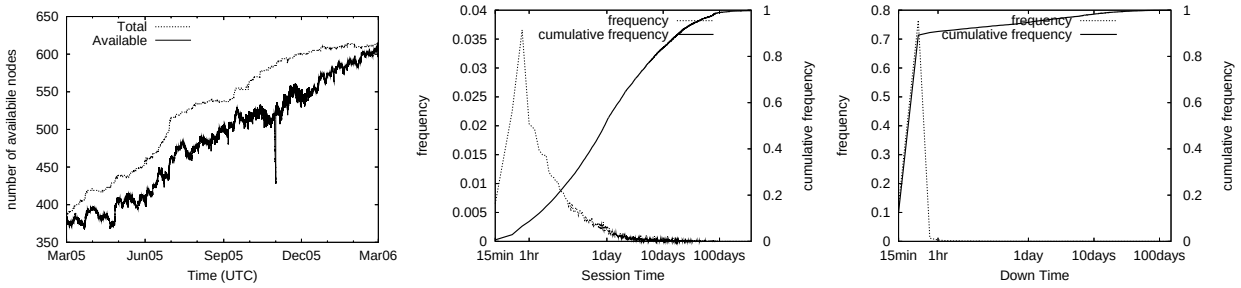
Figure 2.4: PlanetLab First Interval Trace Characteristics (All-pairs Ping). Permanent server failure interarrival distribution.

higher. However, 22% of the servers are available less than 50% of the time.

Table 2.1 summarizes the permanent and transient failures and time between failures. As shown in Figure 2.4, 25% of the permanent failures occurred simultaneously due to multiple server upgrades and a compromise of the system; the time between 50% of permanent failures was separated by three hours or less. Further, the permanent failure interarrival distribution is long tailed with 10% adjacent failures separated by at least eight days or more.

2.1.3 PlanetLab Second Interval: Insights into Matured System and Operation

Figure 2.5 shows the PlanetLab second interval trace characteristics using the CoMon+PLC data set. In contrast to the first interval, the second interval has similar average sessiontime, 194.0 hours (8.1 days) compared to 204.4 hours (8.5 days). However, the average downtime is shorter, 29.0 hours (1.2 days) compared to 82.8 hours (3.5 days). As a result, the average server availability is higher in the second interval, 0.87 compared to 0.70. Furthermore, in the second interval, 50% of



(a) Number of available servers v Time (b) Sessiontime CDF and PDF (c) Downtime CDF and PDF

Distribution	Mean(d)	Stddev(d)	Median(m)	Mode(m)	Min(m)	90th(d)	99th(d)	Max(d)
Session	8.1	25.9	1305	45	5	19.0	39.7	365.0
Down	1.2	5.7	20	30	15	0.1	1.2	224.9
Availability	0.870	0.127	0.990	1.000	0.000	1.000	1.000	1.000

(d) Sessiontime, Downtime, and Availability Statistics (d=days, m=mins)

Figure 2.5: PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Sessiontime, downtime, and availability distributions.

Dates	1 March 2005 – 28 Feb 2006
Number of hosts	632
Number of transient failures	21255
Number of disk failures	219
Transient host downtime (s)	1208, 104647, 14242
Any failure interarrival (s)	305, 1467, 3306
Permanent failures interarrival (s) (Median/Mean/90th percentile)	54411, 143476, 490047

Table 2.2: PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Permanent and transient server failure distributions.

the servers are available at least 99% of the time and 2% of the servers are available less then 50% of the time.

Table 2.2 summarizes the permanent and transient failures and time period between failures. Notice in Figure 2.4, the shape of the curve for permanent failure interarrival times indicates that an exponential distribution is a reasonable fit. The best fitting exponential distribution uses the maximum likelihood estimation with a mean of 1.7 days (143,893 seconds).

2.1.4 Synthetic trace

We also generated synthetic traces of failures by drawing failure interarrival times from exponential distributions. Synthetic traces have two benefits. First, they let us simulate longer time periods, and second, they allow us to increase the failure density, which makes the basic underly-

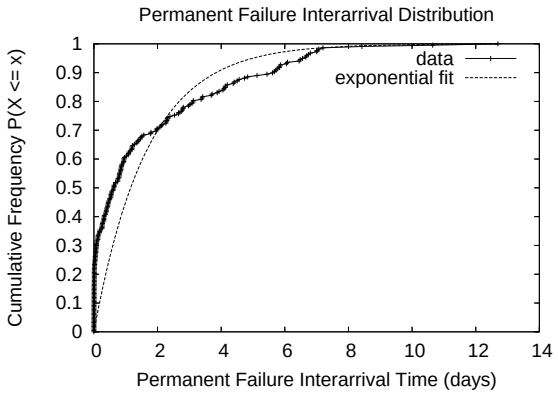


Figure 2.6: PlanetLab Second Interval Trace Characteristics (CoMon+PLC). Permanent server failure interarrival distribution.

ing trends more visible. We conjecture that exponential inter-failure times are a good model for disks that are independently acquired and operated at geographically separated sites. Exponential intervals are possibly not so well justified for transient failures due to network problems.

Each synthetic trace contains 632 servers, just like the PlanetLab second interval trace. The mean sessiontime and downtime match the values shown in Table 2.2 and Figure 2.5.(d). However, in order to increase the failure density, we extended the length to two years and reduced the average server lifetime to one year.

2.2 Analyzing Algorithmic Solutions

This thesis uses a variety of methods to test the effectiveness of its techniques. These methods show that significant efficiency gains can be realized in distributed wide-area storage systems via analysis, trace-driven simulation, and real implementation of algorithms. For instance, in Part II, we explain principled ways for choosing the redundancy type, number of replicas, data placement, and repair strategies. Our methods are based on both analytical models and empirical measurements. We describe our methods in the following sections.

First, we present a unified view of storage algorithms in Section 2.2.1. Next, we describe the metrics we use to measure the effectiveness of each storage algorithmic parameterization in Section 2.2.2. In Section 2.2.3, we describe the subsystem responsible for collecting and analyzing each storage algorithmic solution. Then, in Section 2.2.4, we discuss the criteria to make conclusions about algorithms. Finally, we describe the different environments where the tests are

General Parameters	Description
Redundancy type	Replication or erasure-codes.
Replication level	Number of replicas or fragments required to be available.
Rate of encoding	Fraction of fragments required to read a data object.
Placement strategy	Policy used to select servers to store replicas (e.g. random).
Placement scope	Number of servers eligible to stores replicas for a particular object.
Failure detection time	Timeout used to determine a server is unavailable.

Repair Parameters	Description
Extra replicas	Number of replicas created beyond replication level.
Reintegrate	Reintegrate replicas returning from transient failure.

Table 2.3: Storage System Algorithm Parameterization

Parameter	Dhash	Dhash++	PAST	Pond	TotalRecall	Carbonite
Redundancy type	repl	erasure	repl	erasure	erasure	repl
Replication level	5	14	9	24	variable	3
Rate of encoding	N/A	$\frac{1}{2}$	N/A	$\frac{1}{2}$	$\approx \frac{1}{3}$	N/A
Failure detection time	< 15	< 15	< 15	< 15	< 15	< 15
Placement strategy	Random	Random	Random	Random	Random	Random
Placement scope	5	14	9	N	N	N
Extra replicas	0	0	0	6	10	0
Reintegrate	no	no	no	no	no	yes

Table 2.4: Existing Storage System Parameterization

performed in Section 2.2.5.

2.2.1 Algorithmic Solution Representation

Using a unified view of distributed wide-area storage systems, we can measure and compare the effectiveness of different algorithms. We assume that each algorithm is parameterized based on general system parameters. These parameters are used to maintain durability while reducing costs due to transient failure. We assume further that these parameters are defined once for the whole system and applied individually to each server and data object. The parameters are described in Table 2.3.

As an example of the utility of these parameters, we describe the parameterization of six existing wide area storage systems in Table 2.4: Carbonite [CDH+06], Dhash [DKK+01], Dhash++ [DLS+04], PAST [DR01], Pond [REG+03], and TotalRecall [BTC+04]. These storage systems have been measured and deployed. Furthermore, their parameterizations are described in literature and many other notable storage systems are derived from them. In Part II, we demonstrate

and evaluate the durability and cost of many storage algorithm parameterizations.

2.2.2 Metrics

Storage system algorithmic solution parameterizations are measured and compared based on durability and cost.

Durability is the probability that a data object exists after a specific amount of time. We use many metrics to measure durability. For instance, in Section 4.2, we choose the replication level based on the probability that an object is permanently lost in a particular amount of time. Durability is one minus this probability. Alternatively, in Section 4.1.1, we use a failure rate, number of times a particular (fixed-size) data object fails per unit time. For example, we use the fraction of blocks lost per year (FBLPY) as a failure rate. In the special case – when the likelihood of failure remains constant as time passes such as with an exponential failure distribution – the failure rate is simply the inverse of the mean-time-to-failure (MTTF) for a particular data object.

Cost is the amount of resources used per time to maintain a target durability or replication level. In Chapters 5 and 11 we measure the cumulative number of replicas created. This measures the total bytes sent to maintain a specific durability. Additionally, we measure the bytes sent due to monitoring. However, in many situations the amount of bandwidth used for monitoring is significantly less than the bandwidth due to replica creation.

2.2.3 Statistics Gathering and Analysis

Gathering and analyzing statistics from simulation is immediate. For instance, we count the number of data objects permanently lost to measure durability. On the other hand, we count the number of replicas created to measure cost. Additionally, we measure the data availability since we know the exact time data objects are available or not. Simulation allows us to analyze particular storage algorithms over particular failure traces. This is possible since simulation can measure the percentage of objects lost, availability of objects, and the amount of bandwidth needed to sustain objects over time.

Similar to simulation, we measure object durability and cost in our deployed system. However, the subsystem responsible for gathering and analyzing these statistics is more complex in the deployed system. The subsystem is broken into two parts: event recorder and analyzer.

The event recorder subsystem is run by each server and has four components. First, each server maintains a generation identifier. This is a random number created and stored to disk when the

server is installed or reinstalled following a disk failure. Second, periodically (every minute), a local process on each server probes the local storage application testing if the storage server application is available. Third, each server maintains a database of locally stored object replicas. Finally, each server logs the request and response of every request.

The analysis subsystem is run at a central location and is broken into two components: downloading and analysis. Periodically, once a day, the central site contacts every server downloading their generation identifier, server application availability, list of object replicas, and log of requests. Given the server event logs, the analyzing subsystems measures the number of objects lost, unavailable, and the cost to maintain objects in the system.

The analyzing subsystem has a caveat, it does not know which objects exist or not for each moment. For example, if a server is available but has a new generation identifier, then the central site knows all the object replicas stored on the old generation have been lost. If, however, the central site cannot contact a server, then it does not know whether the object replicas exist on the server or not. In this case, the central site marks the server as unavailable and analyzes the logs accordingly. If the server later becomes available, the event logs are downloaded and the central site uses the new server logs. Since the statistics are produced via logs, the central site simply reruns the analyzing routines over all the downloaded logs.

2.2.4 Algorithmic Solution Comparison Criteria

Ideally, a storage system will not permanently lose any data objects and the cost (number of replicas created) will be proportional to the rate of permanent server failure. Real storage systems, however, are less than ideal. As a result, the effectiveness of an algorithmic solution is based on the difference between the ideal and measured durability and cost. For instance, in Chapter 5, we compare the cost for each algorithm to an oracle that can differentiate permanent and transient server failures and only reacts to permanent failures. Using this method, we show that an algorithm that reintegrates replicas returning from transient failure is nearly optimum. The algorithm initially incurs a penalty for not distinguishing permanent and transient failures, then creates replicas proportional to the permanent failure rate. In general, good algorithmic solutions are measurably closer to an ideal system using an oracle.

2.2.5 Environments

Simulation is the main method used to analyze algorithms. After simulating system behavior, we use the insights gained from simulation to design, implement, and deploy best approaches. The deployments are configurable: we are able to change the parameterization of a deployed system by changing the configuration file for each server.

We currently run two separate deployments discussed in Part IV. Both deployments are parameterized to replicate each object on a configuration of seven storage servers (except where explicitly stated otherwise). Each configuration can tolerate *two* faulty servers and still have a majority. Both deployments are hosted on machines shared with other researchers, and, consequently, performance can vary widely over time.

We apply load to these deployments using 32 servers of a different local cluster. Each machine in the *test cluster* has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory, and two 36 GB disks. 1024-bit RSA signature creation and verification takes an average of 6.0 and 0.6 ms, respectively. The cluster shares a 100 Mbps link to the external network. This cluster is also a shared site resource, but its utilization is lower than the storage cluster.

Simulation

We use the failure traces to drive an event-based simulator. In the simulator, each server has unlimited disk capacity, but limited link bandwidth. However, it assumes that all network paths are independent so that there are no shared bottlenecks. Further it assumes that if a server is available, it is reachable from all other servers. This is occasionally not the case on PlanetLab [FLRS05]; however, techniques do exist to mask the effects of partially unreachable servers [And04].

The simulator takes as input a trace of transient and disk failure events, server repairs and object insertions. It simulates the behavior of servers under different protocols and produces a trace of the availability of objects and the amount of data sent and stored by each server for each hour of simulated time. Each simulation maintains 1 TB of unique data (50,000 data objects, each of size 20 MB). Unless otherwise noted, each server is configured with an access link capacity of 150 KBytes/s, roughly corresponding to the throughput achievable under the bandwidth cap imposed by PlanetLab. The goal of the simulations is to show the percentage of objects lost and the amount of bandwidth needed to sustain objects over time.

Emulation

The first deployment runs on 30 servers of a local cluster run by the Petabyte Storage Initiative [UoC]. Each machine in the *psi cluster* has one 1.0 GHz VIA processor with 1.0 GB of memory, and up to 1 TB of disk space of which we can only utilize 50 GB due to resource sharing. Servers are connected via a gigabit Ethernet switch. 1024-bit RSA signature creation and verification routines take an average of 12.5 and 0.7 ms, respectively. This cluster is a shared site resource; a load average of 10 on each machine is common.

Real Deployment

The other deployment runs on the *PlanetLab* distributed research test-bed [BBC+04]. We use 400+ heterogeneous machines spread across most continents in the network. While the hardware configuration of the PlanetLab servers varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. 1024-bit RSA signature creation and verification take an average of 8.7 and 1.0 ms, respectively. PlanetLab is a heavily-contended resource and the average elapsed time of signature creation and verification is often greater than 210.5 and 10.8 ms.

2.3 Discussion

To test solutions proposed in this thesis we often utilize PlanetLab [BBC+04], a surprisingly volatile environment [PFM06]. Even though it is not typical for current enterprise storage systems, an environment such as PlanetLab's—composed of servers from multiple autonomous organizations that are geographically dispersed—may be more common for many new distributed systems such as the Global Information Grid (GIG) [Age] and GRID. In these new distributed systems, servers cooperate across the wide-area to provide services such as persistent storage. Systems designed in this manner exhibit good scalability and resilience to localized failures such as power failures or local disasters. Unfortunately, distributed systems involving multiple, independently-managed servers suffer from new challenges such as security (including malicious components), automatic management (reliable adaptation to failure in the presence of many individual components), and instability. In PlanetLab, for example, typically less than half of the active servers are stable (available for 30 days or more) [PFM06].

Providing secure, consistent, and available storage in these systems that exhibit extremely

high levels of churn, failure, and even deliberate disruption is a challenging problem and the subject of this thesis. Demonstrating techniques, designs, and implementations that operate well in these environments is a contribution that will lend itself to other distributed wide-area storage system endeavors. This thesis, however, does not investigate peer-to-peer environments composed of home users since bandwidth is not sufficient to durably store data [BR03]; instead, we focus on professionally managed environments where bandwidth is sufficient.

Part II

Maintaining Data Durability Through Fault Tolerance and Repair

Chapter 3

Fault Tolerance and Repair Overview

Wide-area distributed storage systems typically use replication to provide two related properties: durability and availability. *Durability* means that objects stored in the system are not lost due to permanent server failure such as disk failure; whereas *availability* means that the system will be able to return the object promptly. Objects, self-contained units of storage, can be durably stored but not immediately available. If the only copy of an object is on the disk of a server that is currently powered off, but will someday re-join the system with disk contents intact, then that object is durable but not currently available.

The threat to durability is losing the last copy of an object due to permanent server failure. We assume all content stored by a server is lost when it permanently fails. Efficiently countering this threat to durability involves two techniques: fault tolerance and repair. Fault tolerance ensures that data is durable despite permanent failure. It is characterized by an object's configuration, which defines the type of redundancy (replication or erasure-codes), number, and location of replicas. For example, creating three replicas of an object via replication and placing each replica on a disk that fails independently of each other is a fault tolerant technique that can tolerate two failures. Fault tolerant techniques do not respond to server failure; rather, they are designed to tolerate them. Since any particular fault tolerant technique can only tolerate a finite number of failures, redundancy lost due to permanent failure must be replaced eventually; otherwise, given time and failure, the object would be permanently lost. Repair is the process that replaces redundancy lost to permanent failure.

This part of the thesis explores fault tolerant and repair algorithms designed to durably store objects and at a low bandwidth cost in a system that aggregates the disks of many servers distributed throughout the wide-area. In particular, we seek to answer two questions. First, given a set of fault tolerant and repair algorithms what is the associated durability measured in percent of

objects lost per unit time? Second, what is the cost of the algorithm as measured by replicas created over time? Below, we highlight some of the insights discussed in Chapters 4 and 5.

Fault tolerance is the first key to ensuring data durability. The goal of fault tolerant algorithms is to tolerate permanent disk failure without permanent object loss. The storage system must choose the type of redundancy. Both replication and erasure-codes duplicate data in order to reduce the risk of data loss and are considered redundancy. The type of redundancy, however, is more of a designer decision since replication is simple but expensive (bandwidth and storage overhead) and erasure-codes are more efficient but complex. In this thesis, we use the term replica to refer to both replication and erasure-code schemes unless otherwise noted. In addition to the redundancy type, the storage system must choose how much replication to use; the proper amount is related to the burstiness of permanent failures. Finally, the storage system must also choose where to store replicas. Less time is required to recover from failure when replica sets for different data objects are spread over many servers thus allowing more servers to assist in repair. The decrease in repair time increases durability since durability is inversely proportional to repair time [PGK88].

Repair is the other key to ensuring data durability. The goal of repair is to refresh lost replicas before data is lost due to permanent failures. Monitored information, which measures the number of available replicas, is the basis for initiating repair. However, this monitored information is imprecise since replicas can be durably stored but not immediately available, hence transient failure. Initiating repair after failure, whether permanent or transient, is the method currently used by most existing systems. This method may serve as a solution but proves to be costly since creating replicas in response to transient failure is not necessary to maintain durability. We demonstrate how to minimize repair cost. The solution requires many replicas to be simultaneously unavailable before repair is initiated and depends on whether data is mutable or not. For mutable data, servers that return from failure need to either be updated or removed from the replica set if a write occurred while the server was unavailable. To reduce repair cost, the system must estimate the number of replicas that are required to be simultaneously unavailable before repair is initiated. For immutable data, however, reintegrating replicas from transient failures into replica sets minimizes the number of copies created incorrectly due to transient failures. The result is that the system performs some extra work for each object early in its life, but over the long term creates new copies of the object only as fast as it suffers permanent failures.

In the rest of this chapter we present an example for maintaining data durability that will be used in the following chapters. We begin with a set of assumptions and model of the system. Next, we present the redundancy and reconstruction mechanisms.

3.1 System Model

Fault tolerance and repair algorithms, along with a data object *replica location and repair service* work together to maintain data durability.

A replica location and repair service is used to locate and monitor data object replicas and trigger a repair process when necessary. An implementation affects the possible policies for fault tolerance and repair. It knows for each object the location of each replica and number of total and remaining[1] replicas; therefore, it knows how to resolve object location requests and when to trigger repair. The replica location and repair service abstraction gives the storage system location independence and allows different fault tolerance algorithms to be implemented. It may be a central service where a database records the location of all object replica locations and triggers repair if replicas are unavailable; alternatively, it may be a distributed directory where responsibility is partitioned among the servers.

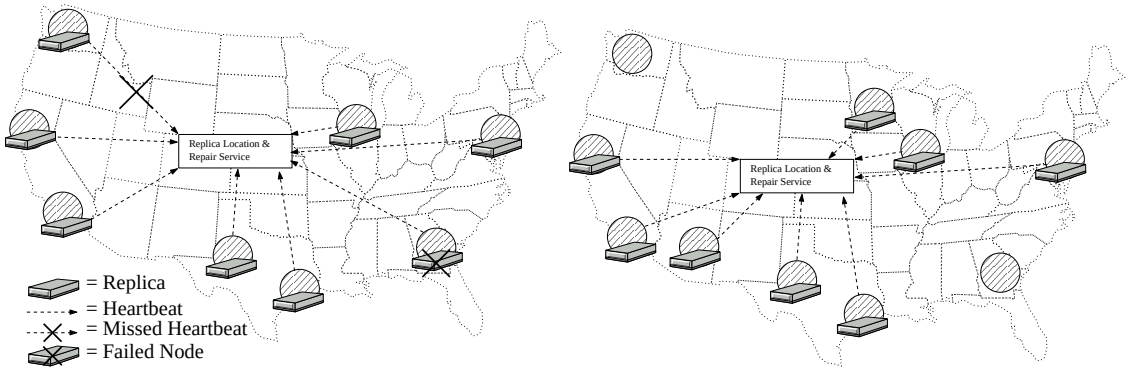
Fault tolerance algorithms choose a configuration that defines the type of redundancy, number of replicas, and placement of each replica. It is invoked when an object is initially inserted into the storage system and when an object is repaired.

Repair works by maintaining a *low watermark* r_L on the number of replicas. When the number of replicas falls below the r_L, the replication level is increased at least back to r_L (some repair algorithms increase the replication level higher [BTC+04]). The data object replica location and repair service tracks the number of available replicas and triggers repair when the available replicas is less than the low watermark.

3.2 Example

Consider the following example to understand the interaction between fault tolerance, repair, and the replica location and repair service. Initially, a fault tolerance scheme uses a replication redundancy algorithm to produce eight total replicas for some object with a low watermark r_L of seven replicas required to be available to satisfy some data durability constraint. Using a data placement strategy, replicas are then distributed throughout the wide-area. Over time, suppose that, a replica in Georgia permanently fails losing data and a replica in Washington transiently fails when a heartbeat is lost (Figure 3.1(a)). As a result of the failures, a repair process might create two new replicas, one in Arizona and the other in Minnesota (Figure 3.1(b)). Repair uses a fault tolerance

[1] The number of remaining replicas is the number of replicas that reside on servers that are currently available.



(a) Threshold Reached (b) Data Recovered

Figure 3.1: Example of Maintaining Data in a Wide Area Storage System.

scheme to choose new servers to store the new replicas.

One might ask what if the server in Washington returns from failure with data intact? There are two possible solutions. First, the system could *reintegrate* the extant replica into the replica set. Reintegrating the extant replica reduces the probability of triggering repair due to future transient failures since the replica set is larger. In the above example, reintegrating the Washington replica into the replica set would result in nine instead of eight total replicas; as a result, three replicas instead of two would have to simultaneously fail for a subsequent repair to be triggered. If the returning replica needs to be updated, however, the benefits of reintegration can be reduced. Systems that do not allow object updates – such as systems that store immutable (read-only and cannot change) data objects – retain the full benefits of reintegration. On the other hand, the system could *forget* about the returning replica. If the object was mutable and an update occurred while the Washington replica was unavailable, then reintegrating the server may actually increase costs since the replica needs to be updated.

In this part of the thesis we investigate fault tolerance algorithms, when to trigger repair, and how to perform repair.

Chapter 4

Fault Tolerance

Fault Tolerance is the first key to ensuring durability. It is defined by a configuration that includes the type of redundancy, number of replicas, and location of replicas. It also represents a data object's ability to tolerate permanent server failure without permanent data loss where no replicas exist anywhere. In Section 4.1, we demonstrate that the type of redundancy is more of a designer decision since replication is simple but expensive (bandwidth and storage overhead) and erasure-coding is more efficient but complex. Next, in Section 4.2, we show the proper number of replicas to create is related to the burstiness of permanent failures. Finally, in terms of replica placement in Section 4.3, we show that less time is required to recover from failure when replica sets for different data objects are spread over many servers. This allows more servers to assist in repair. The decrease in repair time increases durability since durability is inversely proportional to repair time [PGK88]. Furthermore, we show that a random replica placement strategy – such as one that avoids blacklisted servers and replaces duplicate sites – is sufficient to avoid the problems introduced by many observed correlated failures.

4.1 Choosing Redundancy Type

Redundancy type is the first parameter that a fault tolerance algorithm must choose. Redundancy is the duplication of data in order to reduce the risk of data loss. We address two categories of redundancy: simple replication and erasure-coding. The limitation with replication is that it increases the storage overhead and maintenance bandwidth without comparable increase in fault tolerance. In particular, a linear increase in the replication level results in only a linear increase in the number of failures that can be tolerated. In contrast, erasure-coding has a better balance between

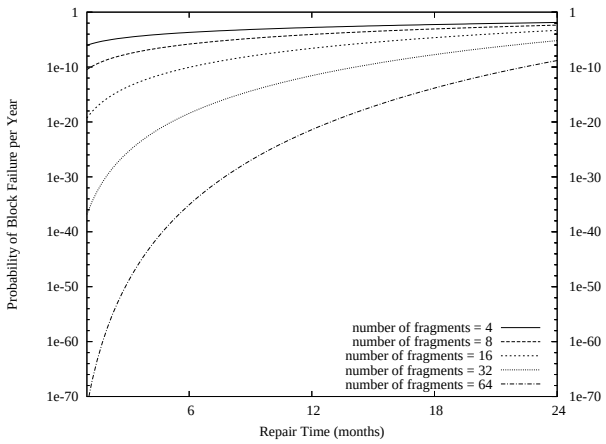


Figure 4.1: Fraction of Blocks Lost Per Year (FBLPY) for a rate $\frac{1}{4}$ erasure-encoded block. Disks fail after five years and a repair process reconstructs data regularly. The four-fragment case (top line) is equivalent to simple replication on four servers. Increasing the number of fragments increases the durability of a block while total storage overhead remains constant. Notice, for example, that for a repair interval of 6 months the four-way replication (top line) loses 0.03 (3%) blocks per year while the 64 fragments, any 16 of which are sufficient to reconstruct (bottom line) loses 10^{-35} blocks per year.

storage overhead and fault tolerance. With a linear increase in storage, the number of server failures tolerated often increases exponentially (e.g. when the number of fragments required to reconstruct the object is greater than one).

Erasure-coding provides redundancy without the overhead of strict replication[CEG+96, DFM00, RWE+01, WK02]. There are three types of erasure-codes: optimal, near optimal, and rateless. Optimal erasure-codes such as Reed-Solomon [BKK+95, Pla97, RV97] encode an object into n *fragments*, any m of which are sufficient to reconstruct the object ($m < n$). We call $r = \frac{m}{n} < 1$ the *rate* of encoding. A rate r optimal erasure-code increases the storage cost by a factor of $\frac{1}{r}$. For example, an $r = \frac{1}{4}$ encoding might produce $n = 64$ fragments, any $m = 16$ of which are sufficient to reconstruct the object, resulting in a total overhead factor of $\frac{1}{r} = \frac{n}{m} = \frac{64}{16} = 4$. Note that $m = 1$ represents strict replication, and RAID level 5 [PGK88] can be described by ($m = 4$, $n = 5$). Unfortunately, optimal erasure-codes are costly (in terms of memory usage and/or processor time) when m is large, so near optimal erasure codes such as Tornado codes [LMS+97, LMS+98] are often used. These near optimal erasure-codes require $(1+\varepsilon)m$ fragments to recover the data object. Reducing ε can be done at the cost of increasing processor time. Alternatively, rateless erasure codes such as LT [Lub02], Online [May02], or Raptor [Sho03], codes transform a data object of m fragments into a practically infinite encoded form. We assume the use of optimal erasure-codes

unless otherwise noted.

Weatherspoon and Kubiatowicz illustrated that increased fragmentation provides greatly increased durability[WK02] as shown in Figure 4.1. More importantly, erasure-resilient systems use up to an order of magnitude less bandwidth than replicated systems to provide the same level of data availability and durability[BSV03, BR03, WK02]. However, there are negative consequences to using erasure-codes.

Erasure-codes and their checksum[WWK02] are processor intensive to produce. For example, the Pond [REG+03] prototype performance was limited by erasure-coding; producing erasure-encoded fragments contributed more than 72% (566.9 ms out of 782.7 ms for a 2MB update) of the write latency. As a result, there is a tradeoff between CPU costs and networking efficiency when considering the use of erasure-codes or replication. Additionally, if a metadata layer individually accounts for each fragment, the layer can be overloaded with location pointers. In order to prevent this problem, fragments need to be aggregated together (e.g. extents) so their individual cost can be amortized. Furthermore, repair is more complicated with erasure-codes since a complete data object would have to be reconstructed to produce a new fragment for repair. This repair read adds extra complexity and cost to the erasure encoded system. However, reconstructed data objects can be cached to reduce costs (eliminate read requirement) of subsequent repairs.

We compare replication and erasure-coding more deeply next.

4.1.1 Erasure-coding versus Replication

In this subsection we demonstrate that systems based on erasure-codes use up to an order of magnitude less bandwidth and storage than replication for systems with similar *mean time to data loss* (MTTDL). Furthermore, we show that employing erasure-codes increases the MTTDL of the system by many orders of magnitude over simple replication with the same storage overhead and *repair* policy. For the following discussion, we assume a simple repair model where lost redundancy is periodically refreshed. Later, in Chapter 5, we will show how to perform a more sophisticated repair. We present this system model next followed by a comparison of systems based on replication and erasure-codes.

System Model

In this section, we make several simplifying assumptions to compare the durability and overhead (storage and bandwidth) of systems that use erasure-coding versus replication. First, we

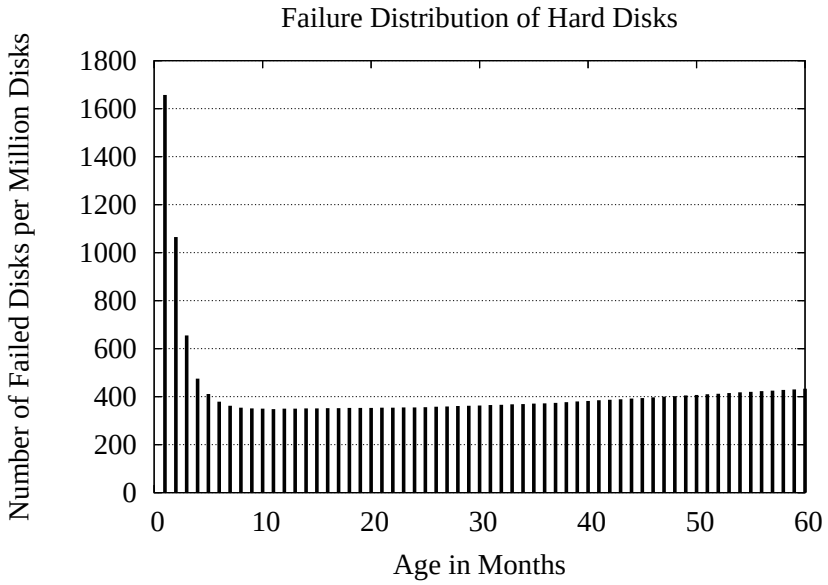


Figure 4.2: Disk Mortality Distribution [PH02].

assume the utilization of a collection of *independently, identically distributed* failing disks—the same assumption made by both *A Case for RAID*[PGK88] and disk manufacturers. Further, we assume that failed disks are immediately replaced by new blank ones[1]. During initial placement or repair, each replica (or *fragment*) for a given data object is placed on a unique, randomly selected disk. Finally, we postulate a global sweep and repair process. The process scans the system, attempting to restore redundancy by reconstructing each data object and redistributing lost replicas (or fragments) over a new set of disks. Notice that repair in *RAID*[PGK88] is *triggered* when a disk fails, which is fundamentally different than sweep and repair. We consider storage systems that trigger repair in Section 4.2. Some type of repair is required; otherwise, data would be lost eventually regardless of the redundancy. See Figure 4.2 for a typical distribution of disk mortality. We denote the time period between sweeps of the same data object an *epoch*.

We perform three comparisons. First, we fix both the *mean time to data loss* (MTTDL) of the system and the length of the *repair epoch*. Second, we fix the storage overhead and repair epoch. Finally, we fix the MTTDL of the system and the storage overhead. Note that the *fraction of blocks lost per year* (FBLPY) illustrated in Figure 4.1 is a rate of object lost and is independent of the size of the system; whereas, MTTDL is the expected time to loss of any object, thus MTTDL of the system is dependent on the MTTDL of an individual object and the system size.

[1]We are ignoring other types of failures such as software errors, operational errors, configuration problems, etc.

For our analysis, we will use the following. We characterize an erasure-encoded or replicated system (denoted by x) in terms of total storage S_x, total bandwidth (leaving the source or entering the destination) BW_x, and the total number of disk seeks required to sustain rate (repair, write, and read) D_x. When comparing the two systems, we will assume that they store the same amount of unique data measured in number of objects, where all objects are the same size. Additionally, new objects are added to the system at a constant rate w (e.g. new data objects per second). We do not compare reads when considering storage and bandwidth because the amount of bandwidth required to read a data object is the same for both systems. We assume the use of optimal erasure-codes such as Reed-Solomon where the object can be reconstructed from m fragments and m fragments is equivalent to one replica in storage and bandwidth requirements.

Fix MTTDL and Repair Epoch

Given a system size defined by the total number of data objects O, we focus on answering the question, what are the resources required to store data in a system long-term? We define the notion of strong durability to be the expected MTTDL of losing *any* data object is sufficiently larger than the expected lifetime of the system. That is

$$MTTDL_{\text{system}} = \frac{MTTDL_{\text{data object}}}{O} \gg \text{system lifetime}$$

We are concerned with the usage of three different resources to maintain strong durability: storage, bandwidth, and disk seeks. The resources can be derived as follows:

$$\begin{aligned} S_x &= \text{total bytes stored in system } x \\ BW_x &= BW_{x_{\text{write}}} + BW_{x_{\text{repair}}} \\ D_x &= D_{x_{\text{write}}} + D_{x_{\text{repair}}} \end{aligned}$$

Where S_x is the total storage capacity required of the system x (where x is replication or erasure-codes), BW_x is a function of the bandwidth required to support both writes and repair of the total storage every repair epoch, and D_x is the number of disk seeks required to support repair, writes, and reads. The repair bandwidth is a function of the total bytes replaced due to server failure during a repair epoch and the length of a repair epoch. We further derive the storage bandwidth, and disk seeks.

First, we compute the storage for both systems

$$\begin{aligned} S_{repl} &= o \cdot R \cdot O \\ S_{erase} &= \frac{o}{m} \cdot n \cdot O = o \cdot \frac{1}{r} \cdot O \end{aligned}$$

Where R is the number of replicas, $r = \frac{m}{n}$ is the rate of encoding, and o is the data object size. R and $\frac{1}{r}$ are the storage overhead factors. Thus, storage is dependent on the data object size, storage overhead factor, and number of data objects.

Next, we show that bandwidth and disk seeks can be expressed in terms of a *DataRate*, the number of original data objects written and repaired.

$$DR_x = N \cdot w + \frac{O \cdot P(e_x)}{e_x}$$

Where the first term represents new writes and includes the number of users N and the new data object creation rate w. The second term represents storage being repaired, where e_x is the repair epoch length, $P(e_x)$ is the probability that an object is lost during the epoch, and $O \cdot P(e_x)$ is the number of data objects that needs repair in system.

Bandwidth is the DataRate multiplied by the size of each object (denoted by o) and storage overhead factor R or $\frac{1}{r}$. Additionally, an erasure encoded system needs to reconstruct a complete data object

before creating a new fragment. Bandwidth can be derived as follows

$$\begin{aligned} BW_{repl} &= BW_{repl_{\text{write}}} + BW_{repl_{\text{repair}}} \\ &= o \cdot R \cdot N \cdot w + \frac{o \cdot R \cdot O_{repl} \cdot P(e_{repl})}{e_{repl}} \\ &= o \cdot R \cdot \left(N \cdot w + \frac{O_{repl} \cdot P(e_{repl})}{e_{repl}} \right) \\ &= o \cdot R \cdot DR_{repl} \end{aligned}$$

$$\begin{aligned} BW_{erase} &= BW_{erase_{\text{write}}} + BW_{erase_{\text{repair write}}} + BW_{erase_{\text{repair read}}} \\ &= o \cdot \frac{1}{r} \cdot N \cdot w + \frac{\frac{o}{m} \cdot n \cdot O_{erase} \cdot P(e_{erase})}{e_{erase}} + \frac{\frac{o}{m} \cdot m \cdot O_{erase} \cdot P(e_{erase})}{e_{erase}} \\ &= o \cdot \frac{1}{r} \cdot N \cdot w + \frac{\frac{o}{m} \cdot O_{erase} \cdot P(e_{erase})}{e_{erase}} \cdot (n+m) \\ &= o \cdot \frac{1}{r} \cdot N \cdot w + \frac{o \cdot \frac{1}{r} \cdot O_{erase} \cdot P(e_{erase})}{e_{erase}} \cdot (1+r) \\ &= o \cdot \frac{1}{r} \cdot \left(N \cdot w + \frac{O_{erase} \cdot P(e_{erase})}{e_{erase}} \cdot (1+r) \right) \\ &= o \cdot \frac{1}{r} \cdot \left(DR_{erase} + \frac{O_{erase} \cdot P(e_{erase})}{e_{erase}} \cdot r \right) \\ &= o \cdot \frac{1}{r} \cdot DR_{erase} \cdot \left(1 + \frac{\frac{O_{erase} \cdot P(e_{erase})}{e_{erase}}}{DR_{erase}} \cdot r \right) \end{aligned}$$

The bandwidth for replication is straightforward and dependent on the write and repair rates multiplied by the size of each data object and storage overhead factor. Bandwidth for erasure-codes is similar to replication except there is an extra term for reconstructing data objects due to repair. The added cost of this extra repair read term is dependent on the ratio of new writes ($N \cdot w$ in the DataRate DR_{erase}) to repair writes $\left(\frac{O \cdot P(e_{erase})}{e_{erase}} \right)$. If new writes dominate repair writes, then the repair read term becomes less significant; otherwise, repair reads reduce the benefit of erasure codes.

Third, we compute the number of disk seeks required to support writes and repair.

$$\begin{aligned} D_{repl} &= R \cdot N \cdot w + R \cdot \frac{O \cdot P(e_{repl})}{e_{repl}} \\ D_{erase} &= n \cdot N \cdot w + n \cdot \frac{O \cdot P(e_{repl})}{e_{erase}} \end{aligned}$$

The above equation states that the number of disk seeks required is dependent on the number of

replicas (or total number of fragments), throughput, system size, repair epoch, and the number of replicas (or fragments) needed to reconstruct the data object.

Finally, a replicated system can be compared to a similar erasure encoded system with the following bandwidth, storage, and disk seek ratios

$$\frac{S_{repl}}{S_{erase}} = R \cdot r \tag{4.1}$$

$$\frac{BW_{repl}}{BW_{erase}} = \frac{o \cdot R \cdot DR_{repl}}{o \cdot \frac{1}{r} \cdot DR_{erase}\left(1 + \frac{\frac{Oerase \cdot P(eerase)}{eerase}}{DR_{erase}} \cdot r\right)} = \frac{r}{1+r} \cdot R \rightarrow R \cdot r \tag{4.2}$$

(As writes begin to dominate repairs)

$$\frac{D_{repl}}{D_{erase}} = \frac{R \cdot DR_{repl}}{n \cdot DR_{erase}} = R \cdot r \tag{4.3}$$

We make the abstract numbers concrete using the following parameters as appropriate. Bolosky et. al [BDET00] measured that an average workstation produces $35\frac{MB}{hr}$ of data. We associate a workstation with a user and assume there are $N = 2^{24}$ users. With $o = 8$kB size data objects, each user writes $w = 17,676$ new data objects per hour. Further, assume we store $O = 10^{17}$ total data objects and wish for the $MTTDL_{\text{system}} > 1000$ years. Hence, the mean time to failure of a particular data object would need to be 10^{20} years ($MTTDL_{\text{data object}} = 10^{20}$ years). As a consequence of the former parameters and using the analysis described in [RWE+01] and reprinted in Appendix A, we solve for a repair epoch length and number of replicas and rate and compute that $e_{repl} = e_{erase} = 4$ months, $R = 22$ and $r = \frac{32}{64} = \frac{1}{2}$ satisfy above constraints, respectively.

Applying these parameters to equations 4.1, 4.2, and 4.3 we produce the following result

$$\begin{aligned} \frac{S_{repl}}{S_{erase}} &= 11 \\ \frac{BW_{repl}}{BW_{erase}} &= 7.33 \rightarrow 11 \\ \frac{D_{repl}}{D_{erase}} &= 11 \end{aligned}$$

These results show that a replicated system requires up to an order of magnitude more bandwidth, storage, and disk seeks as an erasure encoded system of the same size. Erasure-codes are more complicated to use, however. For instance, a data object needs to be reconstructed before a new fragment can be created for repair.

Fix Storage Overhead and Repair Epoch

The same formulas from above can be used to verify durability of system calculations presented in [CEG+96, RWE+01]. For example, using the parameters above and assuming iid failures, we set repair time of $e_{repl} = e_{erase} = four$ months, $R = two$ replicas[2], and rate $r = \frac{32}{64}$. Both the replicated and erasure encoded systems have the same apparent storage overhead of a factor of *two*. Using Appendix A, we compute the $MTTDL_{\text{data object}}$ of a data object replicated onto two servers as 74 years and the $MTTDL_{\text{data object}}$ of a data object using a rate $\frac{1}{2}$ erasure-code onto $n = 64$ servers as 10^{20} years! It is this difference that highlights the advantage of erasure-coding.

Fix MTTDL and Storage Overhead

As a final comparison, we can fix the MTTDL and storage overhead between a replicated and erasure encoded system. This implies that the storage and bandwidth for writes are equivalent for these two systems. In this case erasure encoded systems must be repaired less frequently, and hence, requires less repair bandwidth.

For example, to devise a system with $O = 1000$ data objects where the expected time to lose any data object is 1000 years ($MTTDL_{\text{system}} = 1000$ years), we would want the expected time to lose a particular data object to be $MTTDL_{\text{data object}} = 10^6$ years. A replicated system could meet the above requirements using $R =$ four replicas and a repair epoch of $e_{repl} =$ one month. An erasure encoded system could meet the same requirements using an $r = \frac{16}{64} = \frac{1}{4}$ erasure-code and a repair epoch of $e_{erase} = 28$ months. As a result, the replicated system uses 28 times more bandwidth than erasure encoded system for repair.

If, instead, the system stores $O = 10^{17}$ data objects (as described in subsection 4.1.1) with the same expected time to lose any data object as above ($MTTDL_{\text{system}} = 1000$ years), then the expected time to lose any data object should be $MTTDL_{\text{data object}} = 10^{20}$ years. Using a factor of *four* storage overhead (like in the previous example), the erasure encoded system meets the requirements using an $r = \frac{16}{64} = \frac{1}{4}$ erasure-code and a repair epoch of $e_{erase} = 12$ months, but a replicated system with $R = 4$ replicas would have to repair all data objects almost instantly and continuously.

Discussion

The previous section presented the advantages of erasure-coding, but there are some caveats as well. We highlight three issues: intelligent buffering, caching, correlated failures.

[2] In section 4.1.1 $R = 22$ to attain the same durability

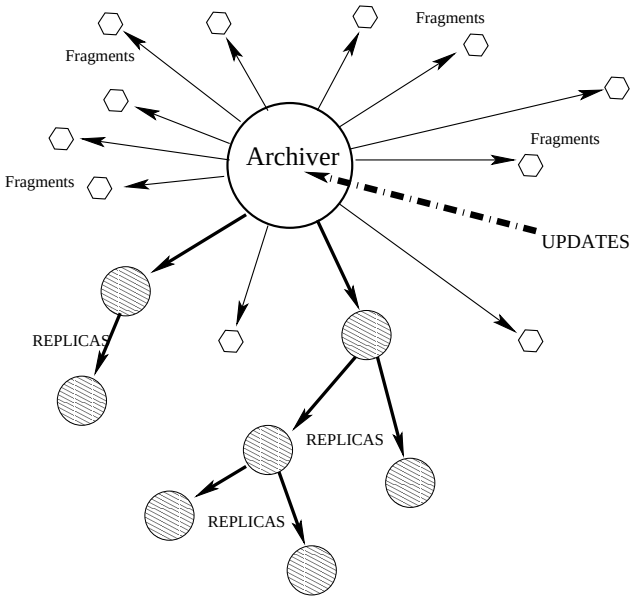


Figure 4.3: Hybrid Update Architecture: Updates are sent to a central “Archiver”, which produces archival fragments at the same time that it updates live replicas. Clients can achieve low-latency read access by utilizing replicas directly.

Each client in an erasure-resilient system sends messages to a larger number of *distinct* servers than in a replicated system. Further, the erasure-resilient system sends smaller “logical” data objects to servers than the replicated system. Both of these issues could be considered enough of a liability to outweigh the results of the last section. We assume two measures could be employed to offset these negative qualities of erasure-coding. First, storage servers can be utilized by a number of clients; this means that the additional servers are simply spread over a larger client base. Second, intelligent buffering and message aggregation can reduce overhead of maintaining many fragments. Although the outgoing fragments are “smaller”, aggregating them together into larger messages and larger disk blocks can reduce the consequences of fragment size. These techniques were implicitly assumed in our exploration via metrics of total bandwidth, storage overhead, and disk seeks in the previous subsection.

Another concern about erasure-resilient systems is that the time and server overhead to perform a read has increased, since multiple servers must be contacted to read a single data object. The simplest answer to such a concern is that mechanisms for *durability* should be separated from mechanisms for *latency reduction*. Consequently, we assume that erasure-resilient coding will be utilized for durability, while replicas (i.e. caching) will be utilized for latency reduction. The advantage of this organization is that replicas utilized for caching are *soft-state* (do not need repair)

and can be constructed and destroyed as necessary to meet the needs of temporal locality. Further, prefetching can be used to reconstruct replicas from fragments in advance of their use. Such a hybrid architecture is illustrated in Figure 4.3. This is similar to what is provided by OceanStore [K+00, REG+03].

Finally, the assumption that failures are *independent* and *identically distributed* is not true in general. Server failures may be correlated because they share network routers, software bugs, configuration problems, operating systems, suffer from same attack, etc.. The rate of failure may be elevated in regions of the network that share common administration or unstable hardware elements. Studies have shown that human errors and network problems are major causes for server failures in the same site [OP02].

We list three possible techniques to address the independence failure assumption. First, greater redundancy gives data a greater chance of surviving correlated failure [HMD05]. Second, sophisticated measurement and modeling techniques could be used to choose a set of servers that are maximally independent during fragment dissemination [jBH+05, WMK02]. Finally, distributing fragments to geographically diverse locations eliminates a large class of correlations caused by natural disasters, denial of service attacks, and administrative boundaries. We show in Section 4.3.2 that random replica placement such as one that avoids blacklisted servers and replaces duplicate sites, is sufficient to avoid the problems introduced by the many observed correlated failures[3].

4.1.2 Complexity of Erasure-Codes and Self-Verifying Data

There are negative consequences to using erasure codes. In particular, erasure codes are more processor intensive to compute than replication and require aggregation and caching to maintain their efficiency. As a result, it is desirable to use complete replication to increase latency performance and erasure codes to increase durability. The challenge is finding synergy between complete replication and erasure coding. Also, maintaining systems built using erasure codes is difficult because erasure coded fragments cannot be verified locally and in isolation, but instead have to either be verified in a group or through higher level objects.

We identify an important challenge when building systems based on erasure codes. In particular, data integrity associated with erasure codes. We contribute a naming technique to allow an erasure encoded document to be self-verified by client and servers. Later, in Part III, we demonstrate how to use this self-verifying property to maintain the integrity of data.

[3]Massively correlated attacks such as virus or worm attack that simultaneously destroy large fractions of the system are out of scope of this thesis.

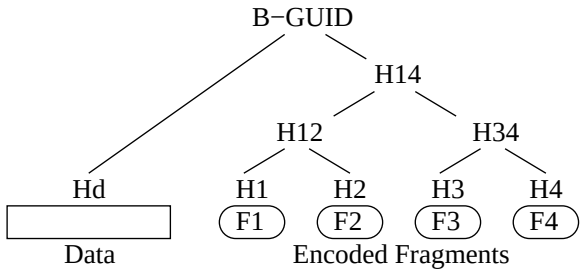
Identifying Erasures: When reconstructing information from fragments, we must discard failed or corrupted fragments (called *erasures*). In traditional applications, such as RAID storage servers, failed fragments are identified by failed devices or uncorrectable read errors. In a malicious environment, however, we must be able to prevent adversaries from presenting corrupted blocks as valid. This suggests cryptographic techniques to permit the verification of data fragments; assuming that such a scheme exists, we can utilize any *m correctly verified* fragments to reconstruct a block of data. In the best case, we could start by requesting *m* fragments, then incrementally requesting more as necessary. Without the ability to identify corrupted fragments directly, we could still request fragments incrementally, but might be forced to try a factorial combination of all returned fragments to find a set of *m* that reconstructs our data; that is, $\binom{n}{m}$ combinations.

Naming and Verification: A dual issue is the *naming* of data and fragments. Within a trusted LAN storage system, local identifiers consisting of tuples of server, track, and block ID can be used to uniquely identify data to an underlying system. In fact, the inode structure of a typical UNIX file system relies on such data identification techniques. In a distributed system with malicious or compromised servers, however, some other technique must be used to identify blocks for retrieval and verify that the correct blocks have been returned. Below we demonstrate that a secure hashing scheme can serve the dual purpose of identifying and verifying both data and fragments. We illustrate how data in both its fragmented and reconstructed forms can be identified with the *same secure hash value*.

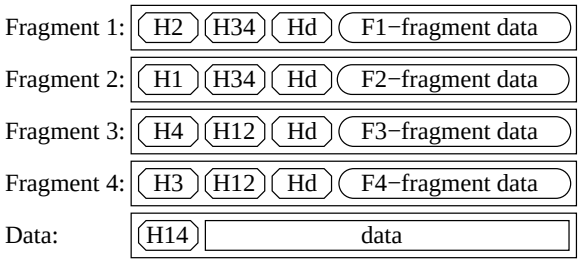
An Erasure Coding Integrity Scheme: We demonstrate how a cryptographically-secure hash, such as SHA-1[NIS94][4], can be used to generate a *single, verifiable name* for data object and all of its encoded fragments.

The scheme works as follows. For each encoded data object, we create a binary verification tree[Mer88] over its fragments and the data object itself as shown in Figure 4.4.(a). The verification tree is produced by computing a hash over each fragment, concatenating the corresponding hash with a sibling hash and hashing again to produce a higher level hash, *etc.*. This process continues until it reaches the topmost hash (H14 in the figure). This topmost hash is concatenated with a hash of the data, then hashed one final time to produce a *globally-unique identifier (GUID)*. The GUID is a permanent pointer that serves the dual purpose of identifying and verifying a block. Figure 4.4.(b) shows the contents of each *verification fragment*. We store with each fragment all of the sibling hashes to the topmost hash, a total of $(\log n) + 1$ hashes, where n is the number of

[4]Other cryptographically-secure hashing algorithms will work as well.



(a) Verification Tree



(b) Verification Fragments

Figure 4.4: (a) Verification Tree: is a hierarchical hash over the fragments and data of a block. The top-most hash is the block's *GUID*. (b) Verification Fragments: hashes required to verify the integrity of a particular fragment.

fragments.

On receiving a fragment for re-coalescing (i.e. reconstructing a data object), a client verifies the fragment by hashing over the data of the fragment, concatenating that hash with the sibling hash stored in the fragment, hashing over the concatenation, and continuing this algorithm to compute a topmost hash. If the final hash matches the GUID for the block, then the fragment has been verified; otherwise, the fragment is corrupt and should be discarded. Should the infrastructure return a complete data block instead of fragments (say, from a *cache*), we can verify this by concatenating the hash of the data with the top hash of the fragment hash tree (hash H14 in Figure 4.4) to get the GUID. Data supplemented with hashes as above may be considered *self-verifying*.

Other erasure-coding verification schemes have been proposed. However, all schemes are significantly more complex and expensive than simple replication. For example, a verification scheme for rateless erasure-coding has been proposed [KFM04]. The advantage of rateless erasure-coding is each fragment produced during repair is unique (with high probability) from all other fragments that exist. Whereas, in (near) optimal erasure-coding, fragments produced during repair may be duplicate of other fragments that already exist. The latter system must carefully track the unique fragments that exist to avoid creating duplicate fragments during repair. Another verification

scheme associates a cryptographically signed certificate with each fragment that includes a secure hash of all fragments (e.g. Glacier [HMD05]). The problem is creating and verifying signatures is significantly more expensive then hashes.

More Complex Objects: More complex structures may be constructed from self-verifying data objects. For example, placing GUIDs into blocks and encoding them is a building block to construct complex objects. The resulting structure is a tree of blocks, with original data at the leaves. The GUID of the topmost block serves much the same purpose as an inode in a file system, and is a verifiable name for the whole complex object. We verify an individual data block (a leaf) by verifying all of the blocks on the path from the root to the leaf. Although composed of many blocks, such a complex object is immune to substitution attacks because the integrity and position of each block can be checked by verifying hashes. Complex objects can serve in a variety of rolls, such as documents, directories, logs, etc. In Part III, we demonstrate how to use very high integrity self-verifying structures such as a secure log.

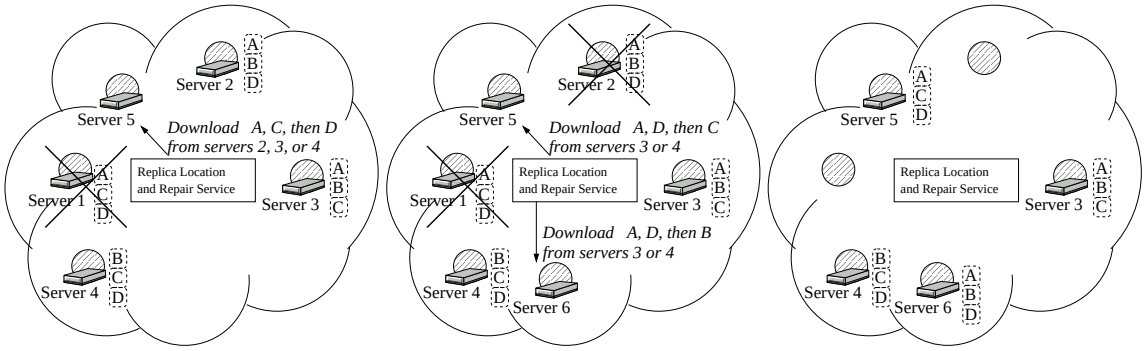
4.2 Choosing the Number of Replicas to Create

The second parameter that a fault tolerance algorithm must choose is the replication level. The choice depends on a target durability level (e.g. probability of data loss after a specific amount of time), distribution of permanent failure bursts, and the ability or rate of creating additional redundancy. Given these parameters, a system designer must choose an appropriate number of replicas to create to meet the target level of durability. In particular, for the characteristics of the system, the number of replicas must be high enough so that the probability of a burst of failures that destroys all replicas is sufficiently rare. Calculating the replication level is the subject of this section.

Replication alone, however, is insufficient to maintain data durability since all servers eventually permanently fail. Over time, permanent server failures decrease the number of replicas that exist. To compensate for this attrition, the system must also use a repair mechanism to create new redundancy to account for lost redundancy. The next Section (4.2.1) discusses the selection of replication level. Additional details of the repair process are discussed in Chapter 5.

4.2.1 System Model

In this section, we assume a replicated system since it is easier to understand and the intuition, derivations, and results are equivalent to an erasure-encoded system. The key difference



(a) Server 1 Fails (b) Server 2 Fails (c) Repair Finished

Figure 4.5: Example repair process. The replica location and repair service coordinates repair process: new servers prioritize downloading new replicas. Initially, there are four servers and four objects (A thru D) with $r_L = 3$ for each object. (a) Server 1 fails. The replica location and repair service selects a new server, server 5, to download lost replicas. (b) Before any repair completes, server 2 fails. The replica location and repair service selects server 6 to download lost replicas and communicates new download priority to server 5 (A,D,C instead of A,C,D since D has less replicas that exist than C). (c) All repair completes. Notice that object replica C on server 5 and B on server 6 waited for an entire servers worth of repairs to complete before they completed.

is that data is lost when no replicas exist in a replicated system or $m-1$ or less fragments exist in an erasure-encoded system. We call the replication level the fault tolerance algorithm selects r_L and study systems that aim to maintain this target level of redundancy in order to survive bursts of failure.

Given a choice of the number of replicas to maintain denoted r_L, the system works as follows. An initial number of replicas, r_L, are stored in the system. We assume that a replica location and repair service monitors each replica and measures the number of replicas that exist over time. When a replica fails due to permanent server failure, a new replica is created. The replica location and repair service selects a new server to host a new replica. The new server downloads the replica from a server storing an existing replica. Finally, the new server updates the replica location and repair service when the download completes.

We assume the new server may have many new replicas to download and prioritizes which replica to download next. Priority is based on the number of remaining replicas that exist for each object. The replica location and repair service communicates the priority to the new server when triggering repair and updates the priority if necessary (e.g. when repair completes by another new server or another failure occurs). The repair time is dependent on the time to download the new replica plus the time to download other replicas with the same or higher priority. Figure 4.5 shows

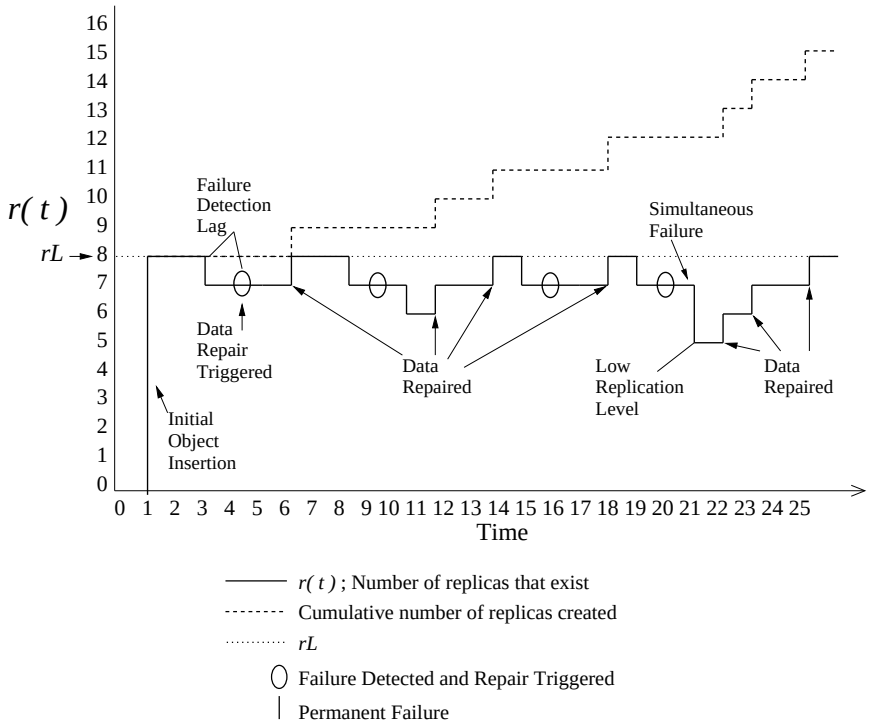


Figure 4.6: Example number of replicas that exist over time. Initially, $r_L = 8$ replicas of an object are inserted into the system. At time 3, a server storing a replica fails. The failure is detected at time 4 and repaired by time 6. The repair lag is due to constrained resources such as access link bandwidth that restrict the number of objects that a server can repair in a given time period. Furthermore, the newly triggered repair would have to wait for previously triggered repairs to complete. Later, another server fails at time 9. But before repair can complete, another server fails at time 11 bringing the number of replicas that exist down to 6. The lost replicas are replaced by time 14. The lowest number of replicas that exist is 5 at time 21.

the process of creating a new replica where repair is coordinated via a replica location and repair service. Details of implementing this service are discussed in Chapter 9.

Replicas are continuously created until one of two situations occurs: either r_L replicas exist again or no replicas exist anywhere and the data object is permanently lost. Figure 4.6 illustrates how the number of replicas that exist evolves over time. Permanent server failures reduce the replication level. Server failure must be detected and repair triggered. The problem is that failure detection has a lag because it takes time to detect failure since wide-area storage systems often use heartbeats to determine that a server is available and lack of a heartbeat to determine a server is unavailable. Further, the repair process has a lag due to constrained resources such as access link bandwidth that restrict the number of objects that can be repaired in a given time period. Moreover, newly triggered repairs may have to wait for previously triggered repairs to complete. We assume

all failures are permanent failures. We study the effects of transient failures, where servers return with data intact, in Chapter 5.

The target replication level r_L is dependent on the failure rate λ_f, distribution of failure bursts, and replica creation rate denoted by μ. The server failure rate λ_f is the average number of times a particular server fails per time unit (assuming that a server can be "renewed" – replaced, after each failure, and then returned to service immediately after failure). In some cases, such as with an exponential distribution, it is the reciprocal of the mean-time-between-failures (MTBF) where MTBF is the average time between failures of a particular server. λ_f (and MTBF) is a system characteristic; see Table 2.2 for example. The replica creation rate is the average number of times one server can copy a particular replica from a remote server per time unit. It is dependent on system characteristics such as the per-server network access link speed, the amount of data stored on each server, and the number of servers (and hence number of access links) which help replace replicas stored on a failed server. When a server fails, new servers must be selected to download replicas from the other remaining servers holding replicas of the objects stored on the failed server. Objects remain durable as long as there is sufficient bandwidth available for the lost replicas to be recreated. For example, in a symmetric system, each server must have sufficient bandwidth to download (and serve) a server's worth of data during its lifetime.

At minimum, if servers are unable to keep pace with the average failure rate, no replication policy can prevent objects from being lost [BR03, CDH$^+$06, Dab05]. These systems are *infeasible*. If the system is infeasible, it will eventually "adapt" to the failure rate by discarding objects until it becomes feasible to store the remaining amount of data. A system designer may not have control over access link speeds and the amount of data to be stored. Fortunately, choice of scheduling which object to repair and object replica placement can improve the speed that a system can create new replicas. Scheduling is considered in our model below and placement is discussed in Section 4.3.1.

If the creation rate is only slightly above the average failure rate, then a burst of failures may destroy all of an object's replicas before a new replica can be made; a subsequent lull in failures below the average rate will not restore the situation since all the replicas are gone. For our purposes, these failures are *simultaneous*: they occur closer together in time than the time required to create new replicas of the data that was stored on the failed disk. Simultaneous failures pose a constraint tighter than just meeting the average failure rate: every object must have more replicas than the largest expected burst of failures. Simultaneous failure due to statistical coincidence is one source of correlation and occurs according to a distribution.

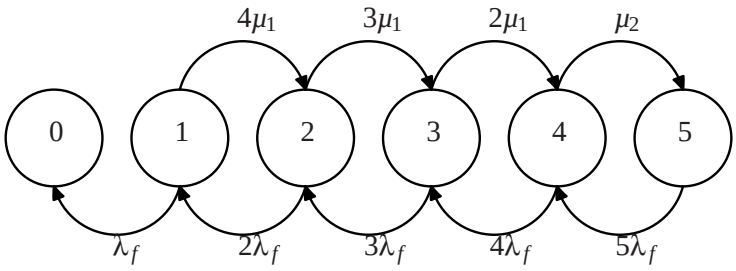


Figure 4.7: A continuous time Markov model for the process of replica failure and repair for a system that maintains three replicas ($r_L = 5$). Numbered states correspond to the number of replicas of each object that exist. Transitions to the left occur at the rate at which replicas are lost; right-moving transitions happen at the replica creation rate.

4.2.2 Generating a Markov Model

We assume that server failure occurs randomly according to an exponential distribution with mean (MTBF) $\frac{1}{\lambda_f}$, where λ_f is the rate of server failure (the number of times a particular server fails per unit time). We assume that λ_f is independent and identically distributed for all servers and that an exponential distribution is reasonable for servers that are independently acquired and operated at geographically separated sites (See Figure 2.6 for example). According to the model presented, an object with i replicas loses a replica with rate $i\lambda_f$ because each of the i servers storing an existing replica might fail.

Further, a replica lost due to server failure is replaced via a new server downloading a new replica from a remote server storing an existing replica. This replica creation process takes some time. For instance, creating a particular replica may first require waiting for an entire servers worth of replicas to be downloaded first. To model repair, we assume that repair occurs randomly according to an exponential distribution with mean $\frac{1}{\mu}$, where μ is the replica creation rate (the number of times one server can download a particular replica from a remote server). We assume that μ is independent and identically distributed for all object repairs and is subject to some randomness due to available bandwidth and competing repairs. An object with i replicas that exist increases the number of replicas with rate $(r_L - i)\mu$. Each of the $r_L - i$ new servers (selected by the replica location and repair service) may complete downloading a new replica.

To analyze the system model presented in this section, we reduce it to a Markov chain. At any point in time, the system has r_i replicas that exist with $(0 \leq r_i \leq r_L)$. The remaining $r_L - r_i$ replicas are lost and need to be repaired (i.e. new server selected and downloads replica from existing servers). An object replication level can be modeled as a Markov chain with $r_L + 1$ states. An object is in state i if there are r_i replicas that exist. In state i, any one of the r_i replicas that exist

can fail, in which case the object goes to state $i-1$. Alternatively, any one of the $r_L - r_i$ non-existing replicas can be repaired, in which case the object goes to state $i+1$. This is a continuous Markov model; in state i, the object moves to state $i-1$ with rate $i\lambda_f$ and to state $i+1$ with rate $(r_L - i)\mu$ where λ_f is the server failure rate and μ the replica creation rate (rate at which a particular server can download a particular replica). We assume objects with lower replication are scheduled for repair with higher priority than objects with greater replication, thus have an increased likelihood of repair. Note that state $i = 0$ is an absorbing state; the object no longer exists and cannot be repaired since no replicas exist anywhere. The model is illustrated in Figure 4.7.

The Markov chain model is useful to compute the probability that a data object exists after some amount of time T given r_L, μ, and λ_f. We discuss this computation in Section 4.2.4. Similar Markov chain model derivations and analyses can be found in literature, Chun et al [CDH+06], Dabek [Dab05], and Ramabhadran and Pasquale [RP06]. The Markov models presented in [CDH+06] and [Dab05] are different, so the results will be different; however, the analytic derivation mechanics are the same. The difference is the rate of creating a new replica is constant, independent of the object's current replication state [CDH+06, Dab05]; whereas, we consider "scheduling" of repair, the likelihood of being selected for repair is increased as more replicas are lost since objects with less replicas are selected with a higher priority. This scheduling is similar to the model presented in [RP06].

4.2.3 Creation versus Failure Rate

Intuitively, the server failure rate λ_f and replica creation rate μ represent a balance between how fast a system loses replicas compared to how fast it can create replicas to compensate for the attrition. This ratio between replica creation rate and server failure rate determines the average replicas per object the system can expect to support [CDH+06, Dab05, RP06]. For example, if the system must handle coincidental bursts of, say, five failures, it must be able to support at least six replicas and hence the replica creation rate must be at least 6 times higher than the replica failure rate. We'll refer to the ratio μ/λ_f as θ. Choices for r_L are effectively limited by θ. It is not the case that durability increases continuously with r_L; rather, when using $r_L > \theta$ the target replication level is greater than the number of replicas that can be created for a particular object per unit time. In this case, the system provides the best durability it can, given its resource constraints (i.e. the average number of replicas will be at most θ instead of r_L when $r_L > \theta$). On the other hand, When $r_L < \theta$, higher values of θ decrease the time it takes to repair an object, and thus the 'window of

vulnerability' during which additional failures can cause the object to be destroyed.

To get an idea of a real-world value of θ, we estimate λ_f and μ based on the historical failure record for permanent server failures on the second interval of the PlanetLab trace. From Table 2.2, the average permanent failure inter-arrival time for the entire test bed is 39.85 hours. On average, there were 490 servers in the system, so we can estimate the mean time between permanent failures for a single server as $490 \cdot 39.85$ hours or 2.23 years. Hence, $\lambda_f \approx 0.439$ permanent server failures per year.

The replica creation rate μ, number of times one server can download a particular replica from a remote server per time unit, depends on the achievable network throughput per server, as well as the amount of data that each server has to store (including replication). We assume that an entire server's worth of data may be downloaded before downloading a particular replica. For example, recall from Figure 4.5 that there were four servers and four objects with r_L = 3 (each server stores 3 replicas). After failure, server 5 downloaded an entire server's worth of replicas before a replica for object C could be downloaded. Similar to this illustration, we estimate μ based on the amount of time to download an entire server's worth of data before downloading a particular replica; As a result, we estimate μ based on the data per server and network access link bandwidth. In PlanetLab, the current limit on the available network bandwidth is 150KB/s per server. If we assume the system stores 500GB of unique data per server with $r_L = 3$ replicas each, then each of the 490 servers stores 1.5TB (Total amount of replicated data is 490·500GB·3=735TB and amount of data per server is 735TB/490=1.5TB). This means that a particular replica, downloaded after a server's worth of data, can be downloaded in 121 days (i.e. 1.5TB/150KBs=121 days), or approximately three times per year. This yields $\mu \approx 3$ per year, one server can download a particular replica three times a year.

Therefore, in a system with these characteristics, we can estimate the ratio between the replica creation rate and server failure rate $\theta = \mu/\lambda_f \approx 6.85$. In practice, this value is somewhat lower; for example, servers cannot make copies during downtimes or shortly after a permanent server failure. However, the fact remains that θ is higher than the minimum for a feasible system ($\theta \geq 1$ defines a feasible system). The system still profits from this because higher values of θ decrease the time it takes to repair an object, and thus the 'window of vulnerability' during which additional failures can cause the object to be destroyed. Furthermore, when viewed in terms of permanent server failures and copies, θ depends on the value of r_L: as r_L increases, the total amount of data stored per server (assuming available capacity) increases proportionally and reduces μ.

To study the impact of θ, we ran a set of experiments via simulation where we reduced the bandwidth per server effectively reducing the replica creation rate μ (and θ). The goal was to

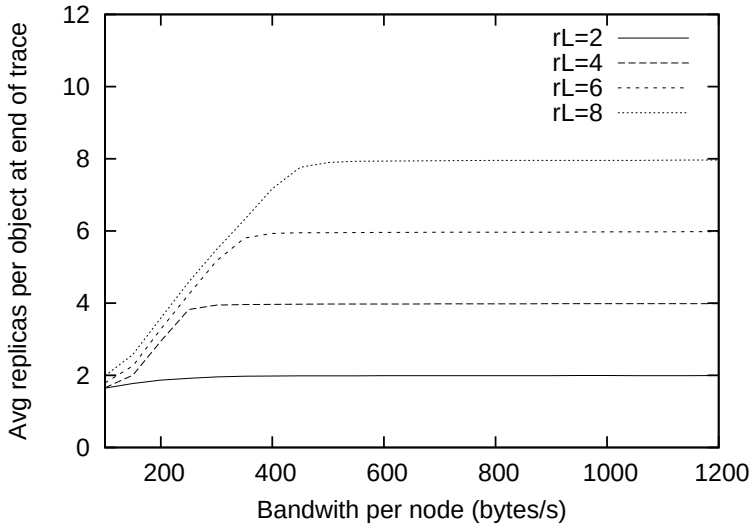


Figure 4.8: Average number of replicas at the end of a two year synthetic trace for varying values of θ. This Figures represents a set of simulations where we reduced the bandwidth per server (x-axis) effectively reducing the replica creation rate μ (and θ). The input to the simulator was a synthetic failure trace with a 632 servers and a server failure rate of λ_f = 1 per year. The storage load maintained was 1TB of unique data (50,000 20MB objects). As a result, the total replicated data was 2TB, 4TB, 6TB, and 8TB for r_L = 2, 4, 6, 8, respectively. Finally, each experiment was run with a specific available bandwidth per server that ranged from 100 B/s to 1,200 B/s.

measure the average number of replicas that exist per object at the end of the trace and relate that to the expected value based on r_L and θ. If $r_L \geq \theta$ then the average should be near θ; otherwise, if $r_L < \theta$, the average should be r_L. The simulator was discussed in Section 2.2.5. In review, the simulator acts as a replica location and repair service. After a server failure has been detected via a timeout, new random servers are selected to download and replace the replicas lost on the failed server. The input to the simulator was a synthetic failure trace with 632 servers and a failure rate of λ_f = 1 per year (Section 2.1.4). The length of the trace was two years. The storage load maintained was 1TB of unique data (50,000 20MB objects). As a result, the total replicated data was 2TB, 4TB, 6TB, and 8TB for r_L = 2, 4, 6, 8, respectively. Finally, each experiment was run with a specific available bandwidth per server which ranged from 100 B/s to 1,200 B/s. For example, the case of 100 B/s corresponds to $\theta = 1.81/r_L$ (i.e. $\mu = \frac{\text{bw}}{\text{data per server}} = \frac{100\text{B/s}}{1TB \cdot r_L/632} = 5.748 \cdot 10^{-8}/r_L$ per sec, $\lambda_f = \frac{1}{1\text{yr}} = \frac{1}{31536000\text{ sec}} = 3.171 \cdot 10^{-8}$ per sec, and $\theta = \frac{\mu}{\lambda_f} = \frac{5.748 \cdot 10^{-8}/r_L}{3.171 \cdot 10^{-8}} = 1.81/r_L$). Figure 4.8 shows the results of these simulations. When $r_L < \theta$, r_L is the average number of available replicas per object at the end of the trace. However, when θ is less than r_L the ratio of the replica creation rate to server failure rate is not sufficient to support an average target replication level of r_L replicas per object. The system can no longer maintain full replication and starts operating in a 'best effort' mode, where higher values of r_L do not give any benefit. The exception is if some of the initial r_L replicas survive through the entire trace, which explains the small differences on the left side of the graph.

4.2.4 Choosing r_L

A system designer must choose an appropriate value of r_L to meet a target level of durability. This process could be automated. Essentially, for the characteristics of the system, r_L must be high enough so that a burst of r_L failures is sufficiently rare.

One approach to choosing r_L would be to simply examine a trace and select one more than the maximum burst of simultaneous failures. For example, Figure 4.9 shows the burstiness of permanent failures in the second interval of the PlanetLab trace by counting the number of times that a given number of failures occurs in disjoint 24 hour and 72 hour periods. If the size of a failure burst exceeds the number of replicas, some objects may be lost. As a result, one may conclude perhaps that 12 replicas are needed to maintain the desired durability. This value would likely provide durability but at a high cost. If a lower value of r_L would suffice, the bandwidth spent maintaining greater numbers of replicas would essentially be wasted.

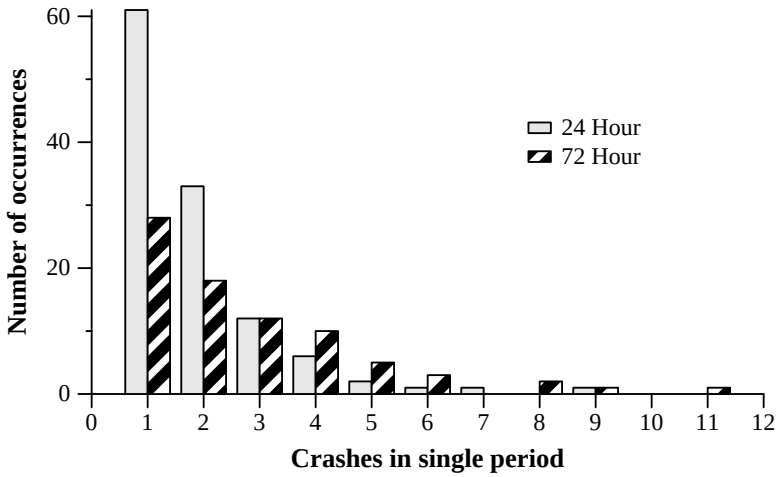


Figure 4.9: Frequency of "simultaneous" failures in the PlanetLab trace. These counts are derived from breaking the trace into non-overlapping 24 and 72 hour periods and noting the number of permanent failures that occur in each period. If there are x replicas of an object, there were y chances in the trace for the object to be lost; this would happen if the remaining replicas were not able to respond quickly enough to create new replicas of the object.

There are several factors to consider in choosing r_L to provide a certain level of durability. First, bursts arrive according to the distribution of failures: there is no maximum burst size. Worse, a burst may arrive while there are fewer than r_L replicas. From this, one could conclude that the highest possible value of r_L is desirable. On the other hand, the simultaneous failure of even a large fraction of servers may not destroy any objects, depending on how replicas are placed. In real systems, the workload may also change over time, affecting μ.

The selection of r_L does not capture the impact of placement strategies: this is best captured via simulations that include real workloads and placement strategies. We discuss placement strategies in Section 4.3.1. The replication level r_L does capture the effect of the burst distribution via the Markov model presented in Section 4.2.1.

Given this Markov model, we can analyze the relationship between the values of r_L, μ, λ_f and the probability that data is lost. By examining the probability of data loss for various combinations of the system parameters, we hope to provide some guidance to system designers who must pick a reasonable value for r_L. Previously we stated that an appropriate value for r_L might be the maximum number of simultaneous failures that the system will experience: since this analysis assumes that failures are independent events, what the calculation here determines is the probability of more than r_L failures occurring due to statistical coincidence.

In the context of the Markov process, the probability that we are interested in corresponds to the probability that the system is in state $i = 0$ at time T. We can find this probability by solving the differential equations that describe the behavior of the Markov process over time. For each state, we can write a differential equation that relates the probability of remaining in a state to the rate of transitions into the state and out of the state. The negative term of each equation captures the transitions out of the state, the positive term captures the in transitions:

$$
\begin{aligned}
\tfrac{dP_0(t)}{dt} &= \lambda_f P_1(t) && \text{// prob enter state } i=0\\
& && \text{// no leaving state } i=0!\\
\tfrac{dP_1(t)}{dt} &= -((r_L-1)\mu+\lambda_f)P_1(t) + && \text{// prob leave state } i=1\\
& \quad 2\lambda_f P_2(t) && \text{// prob enter state } i=1\\
\tfrac{dP_i(t)}{dt} &= -((r_L-i)\mu+i\lambda_f)P_i(t) + && \text{// prob leave state } i\\
& \quad (r_L-i+1)\mu P_{i-1}(t)+(i+1)\lambda_f P_{i+1}(t) && \text{// prob enter state } i\\
\tfrac{dP_{r_L}(t)}{dt} &= -r_L\lambda_f P_{r_L}(t) + && \text{// prob leave state } i=r_L+1\\
&= \mu P_{r_L-1}(t) && \text{// prob enter state } i=r_L+1\\
\\
\textstyle\sum P_i(t) &= 1
\end{aligned}
$$

with the initial conditions $P_{r_L}(0) = 1$ and $P_i(0) = 0$. The final equation stipulates that the probabilities sum to 1; this constraint is necessary to solve the system since the other r_L equations are not independent. This system can be analyzed numerically to estimate the probability of data object loss by time T: $P_0(T)$.

We can analyze the example presented in Section 4.2.3 where 490 PlanetLab servers store 500GB of replicas each with parameters $r_L = 3$, $\lambda_f = 0.439$, $\mu = 3$, and $\theta = 6.85$. Figure 4.10 plots $P(t)$ for each of the four states. The solid bold line shows the probability of object loss over time. The dotted bold line shows the results of a simulation of the same system. The observed fraction of data lost at each point in time is plotted (error bars show the minimum and maximum from five runs of the simulator). The probability that data is lost rises towards one as time increases even though the system can create objects faster than they are lost ($\mu/\lambda_f \approx 6.85 > 1$). If failure events are generated by Poisson processes, object loss is inevitable given enough time, since a burst of r_L failures between repair actions has a nonzero probability.

The shape of the curve in Figure 4.10 is affected mainly by the ratio $\theta = \frac{\mu}{\lambda_f}$. The higher θ, the faster repair is in relation to data loss, so the system spends more time in the nonzero states. Therefore it is important to design the system such that μ is as high as possible. In Section 4.3.1, we

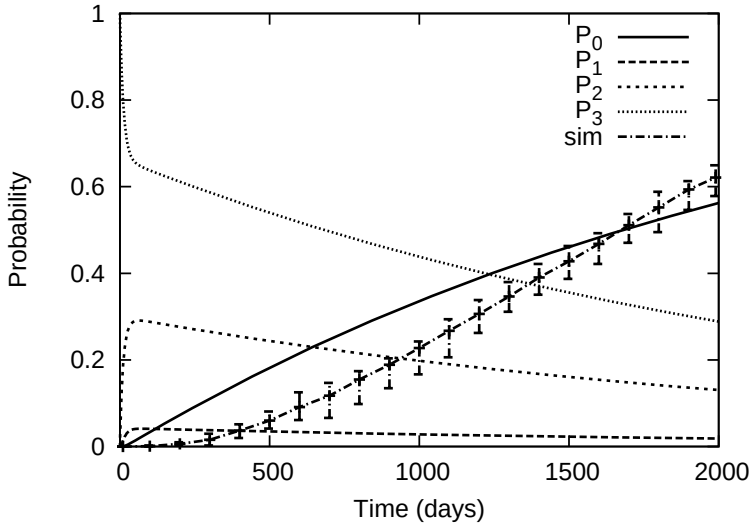


Figure 4.10: Analytic results for the probability of data loss over time. These curves are the solution to the system of differential equations governing a continuous time Markov process that models a replication system running on PlanetLab storing 500GB. At time zero, the system is in state 3 (three replicas) with probability 1.0 (dot-dash line is at 1.0). As time progresses, the system is overwhelmingly likely to be in state 0 which corresponds to object loss (that probability is shown as a bold line in the above plot).

describe placement strategies that can be used to increase μ. The individual values of λ_f and μ also affect the results.

By solving this system of differential equations to determine $P_0(T)$ for various r_L, a designer can estimate an r_L that gives an acceptably small chance of object loss in a bounded time after the object is inserted. To explore different workloads, we consider different amounts of data per server. In Figure 4.11, we graph the probability that an object will survive after four years as a function of r_L and data stored per server which affects the repair rate and hence θ. Note that the amount of unique data stored in the system decreases as r_L increases since we constrain the amount of data per server.

The points plotted are obtained by evaluating the probability that no replicas exist and the object is in state zero after four years $P_0(4years)$ in the continuous time Markov model with r_L +1 total states; each value of r_L requires evaluating a different model. Each curve lowers towards the right: as r_L increases the system can tolerate more simultaneous failures and objects are less likely to be lost. The predicted object loss increases as per-server capacity is increased: when more data must be copied after a failure, the window of vulnerability for a simultaneous failure to occur also

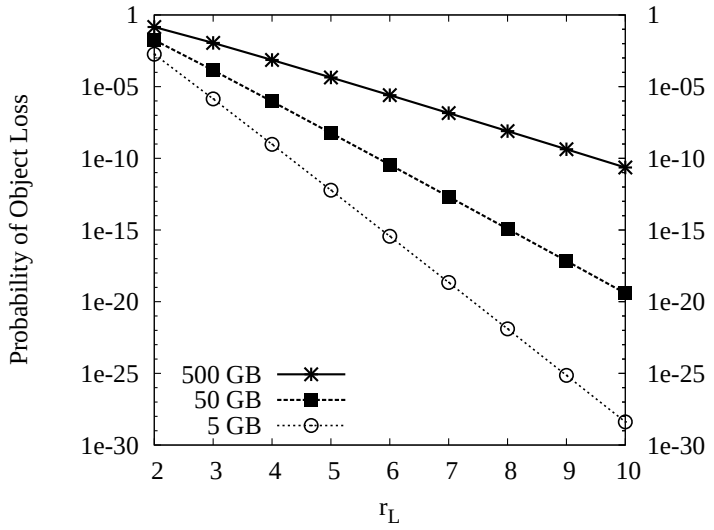


Figure 4.11: Analytic prediction for object durability after four years on PlanetLab. The *x*-axis shows the initial number of replicas for each object: as the number of replicas is increased, object durability also increases. Each curve plots a different per-server storage load; as load increases, it takes longer to copy objects after a failure and it is more likely that objects will be lost due to simultaneous failures.

increases. Finally, $r_L = 1$ is not shown since it corresponds to not using any replication and objects are lost based only on the lifetime of a server (with $r_L = 1$, no new replicas are created, so the object is lost when the only copy is lost).

4.3 Choosing Where to Store Replicas

The third decision a fault tolerance algorithm must consider is the placement of replicas. Replica placement is the process in which servers are selected to store data replicas. There is a plethora of proposed placement algorithms [KKM02] optimizing for properties such as access latency, availability, or load balance. However, in the context of durability, two properties are most relevant: scope and predictability.

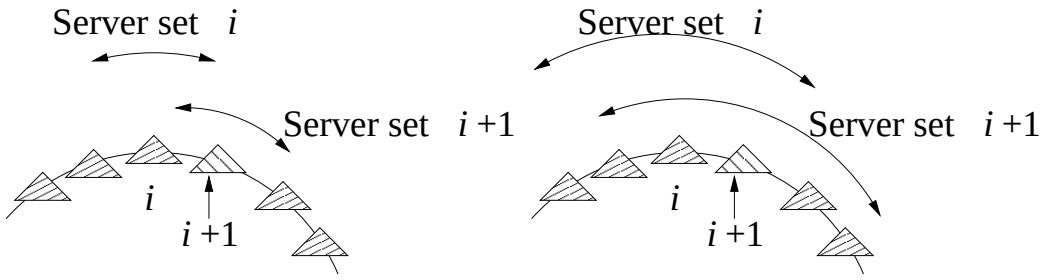
First, *scope* determines which servers are eligible to store replicas for a given object. Furthermore, after replicas are placed, scope implies which servers are monitored for liveness and storage of replicas for particular objects. Systems like GFS [GGL03], Pond [REG+03], and TotalRecall [BTC+04] have a large scope and consider all servers in the system as eligible to store replicas. While storage systems based on distributed hash tables (DHTs) such as Dhash [Cat03],

OpenDHT [RGK+05], and PAST [DR01] to name a few, consider a small number of servers; the exact set depends on the identifier of the object in question. Increasing the scope has two desirable effects: It increases flexibility of the system and it increases the parallelism during repair. In general, with a scope of size k, if one server of the k servers fails, then the $k - r_L - 1$ other servers can use their combined access links to download and replace the replicas lost on the failed server. When scope is small (e.g. $k = r_L$), only a few servers download new replicas (namely, new servers replacing the failed servers download replicas); however, when scope is large (e.g. $k = N$), many servers download new replicas (any server that does not already store the lost replica can download a replacement replica). A large scope can utilize more network paths to replace lost replicas. The disadvantage of a large scope is that higher values of k increase the overhead of monitoring server liveness (monitoring is discussed further in Section 5.1).

Second, a *failure predictor* biases the placement of replicas towards more reliable servers. In other words, failure predictors use an "oracle" to make statements about the expected remaining lifetime of servers or groups of servers. If a predictor is used, the system can improve its chances of maintaining durability by placing replicas on highly reliable groups of servers. Several existing systems use this approach; Pond [REG+03, K+00] infers host reliability from the observed failure pattern [WMK02] while Phoenix [jBH+05] uses server attributes such as operating system or installed software. On the other hand, failure predictors have been criticized as inherently unreliable [HMD05], and measurement studies have shown that failures in real systems are difficult to predict [YNY+04]. When using a failure predictor, there is an obvious tradeoff between load balance and optimized placement. It is only by biasing load towards more reliable servers that durability can be improved; as a consequence, these servers are required to provide more storage (and to answer more requests) than their less reliable counterparts.

Failure predictors can also be used to avoid temporally correlated failures. Temporally correlated failures occur close together in time. A hypothetical failure predictor that knows the exact time servers permanently fail can be used in a placement strategy to avoid temporally correlated failures between pairs or large groups of servers. Avoiding placing replicas on servers that fail at the same time helps ensure there is sufficient time to replace replicas lost to failure. This clairvoyant failure predictor is able to better avoid temporally correlated failures than Weatherspoon et al [WMK02] and Phoenix [jBH+05] since it knows future failure times while the latter strategies do not know the future. We use a clairvoyant failure predictor as a basis for comparison in Section 4.3.2.

As a specific instance of applying scope and failure prediction, we demonstrate two ef-



(a) scope=3 (b) scope=5

Figure 4.12: Scope. Each unique server i has a unique set of servers it monitors and can potentially hold copies of the objects that i is responsible for (server set$_i \not\subset$ storage server set$_j$, $\forall i \neq j$). The *size* of that set is the server's *scope*. (a) scope=3 and (b) scope=5. In terms of placement choices, assuming that r_L =3 and object replicas are stored on server i's server set, then there is no choice for (a) and $\binom{\text{scope}}{r_L} = \binom{5}{3}$ choices in (b).

fects (in the context of systems with failure characteristics such as PlanetLab). First, a large scope increases durability. Second, not much benefit is gained from use of failure predictors since random replica placement – such as one that avoids blacklisted servers and replaces duplicate sites – is sufficient to avoid the problems introduced by many observed correlated failures. Using a failure trace with correlated failures due to upgrade and compromise (first interval of PlanetLab), a random placement policy that avoids blacklisted servers and duplicate sites, triggered only 3.4% more repairs than a clairvoyant placement that knew the future time that servers fail.

We discuss scope and failure prediction further.

4.3.1 Increasing Durability Through Repair Parallelism with Scope

This subsection explores how the system can increase durability by copying objects from a failed server in parallel. In effect, this reduces the time needed to repair the replicas lost on failed servers and increases θ (ratio of rate of replica creation to rate of server failure).

Each server, i, designates a set of other servers that it monitors and can potentially hold copies of the objects that i is responsible for. We call the *size* of that set the server's *scope*, and consider only system designs in which every server has the same scope. Scope can range from a minimum of r_L to a maximum of the number of servers in the system N. Scope and the replica location and repair service (discussed in Section 3.1) are related since scope defines the number of servers a particular server will monitor. Thus, it defines which servers can assist in locating and monitoring servers responsible for storing replicas of a particular object.

To be explicit, each unique server i has a unique set of servers it monitors and can poten-

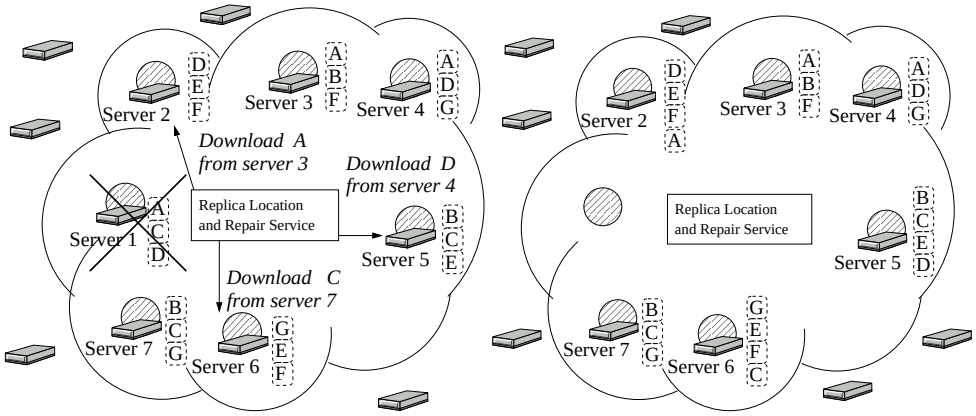
tially hold copies of the objects that i is responsible for (server set$_i \not\subset$ storage server set$_j$, $\forall i \neq j$). All server sets are the same size, and the size of the server set ranges from r_L to N. Each object is replicated within a unique server set. Generally, a larger scope offers more flexibility since the number of possible replica sets is larger (around $\binom{\text{scope}}{r_L}$); however, more servers must be monitored with a larger scope. Figure 4.12.(a) pictorially shows how a server, server set, and replicas for a particular object are organized.

After a particular scope (size of server set) has been chosen, a placement strategy needs to be chosen. For example, if the number of replicas is r_L, scope is N, and placement strategy is random, then any random server may be chosen to store a replica. On the other hand, if the placement is successor list placement (the number of replicas is r_L and scope is N), then only the successive r_L servers would initially be chosen to store replicas, even though all servers are eligible. Notice that successor list placement does *not* imply a small scope (scope is N in this example); rather, successor list placement states that servers successive in the identifier space should initially store replicas. Essentially the placement strategies are restricted to a server's scope. Furthermore, existing copies only "count" towards the replication level of an object if they are stored on one of the servers in the set. We compare placement strategies in Section 4.3.2, but for this section we assume a random placement.

A small scope means that all the objects stored on server i have copies on servers chosen from the same restricted set of other servers. The advantage of a small scope is that it makes it easier to keep track of the copies of each object. For example, DHash stores the copies of all the objects with keys in a particular range on the successor servers of that key range. The result is that those servers store similar sets of objects, and can exchange compressed summaries of the objects they store when they want to check that each object is replicated a sufficient number of times [Cat03, RGK$^+$05].

The disadvantage of a small scope is that the effort of creating new copies of objects stored on a failed disk falls on the small set of servers in that disk's scope. The time required to create the new copies is proportional to the amount of data on one disk divided by scope minus r_L ($\frac{\text{data\_per\_server}}{\text{scope}-r_L}$). A small scope results in a long repair time. Another problem with small scope, when coupled with systems that use consistent hashing [KLL$^+$97, Cat03, DKK$^+$01, DR01, RGK$^+$05], is that the addition of a new server may cause unneeded copying of objects: the small scope may dictate that the new server replicates certain objects, forcing the previous replicas out of scope and thus preventing them from contributing to durability.

Assuming a random placement policy with replicas for a particular object placed in dif-



(a) Server 1 Fails. Coordinate Repair (b) Repair Completes

Figure 4.13: Example parallel repair with a large scope. Scope is 7 and r_L = 3. Only servers within scope are monitored and there are $\binom{7}{3}$ possible replica sets. The replica location and repair service coordinates the repair process utilizing as many source and destination server pairs as possible. Initially, there are seven servers and seven objects (A thru G) with $r_L = 3$ for each object. (a) Server 1 fails. The replica location and repair service selects as many source and destination server pairs to reduce the repair time. Server 2 downloads replica A from server 3. Similarly, server 5 downloads replica D from server 4 and server 6 downloads replica B from server 7. (b) All repair completes.

ferent unique sites, larger scopes spread the work of making new copies of objects on a failed disk over more access links, so that the copying can be completed faster. In the extreme of a scope of N (the number of servers in the system), the remaining copies of the objects on a failed disk would be spread over all servers, assuming that there are many more objects than servers. Furthermore, the new object copies created after the failure would also be spread over all the servers. Thus the network traffic sources and destinations are spread over many network paths, and the time to recover from the failure is short (proportional to the amount of data on one disk divided by N or $\frac{\text{data\_per\_server}}{\text{scope}=N-r_L}$). Figure 4.13 illustrates spreading repair over many sources and destinations server pairs reducing repair time.

A large scope requires coordination to effectively reduce repair time. For instance, in Figure 4.5 and 4.13, a replica location and repair service coordinates many source and destination server pairs to parallelize repair. Many network paths are used in parallel decreasing repair time. In implementation, in Section 9, the replica location and repair service is implemented as a distributed directory. Each server is responsible for monitoring the replicas for particular objects. When a particular object's replication level falls below the target level, the server responsible for monitoring the object triggers repair. The repair process, then, selects a leader to coordinate repair.

Additionally, a larger scope also means that a temporary failure will be noticed by a

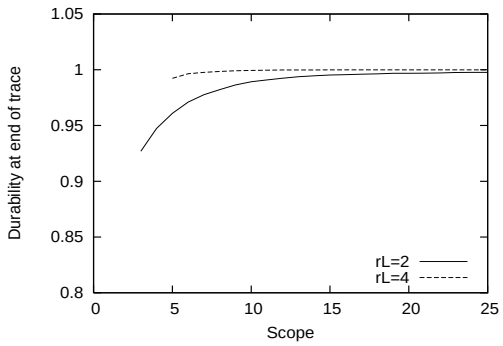


Figure 4.14: Durability for different scopes. Assuming random placement. We vary the target replication level r_L and scope (x-axis). To reduce θ, we limit the bandwidth per server to 1000 B/s in this experiment. Durability is measured via simulation using a two year synthetic trace. Increasing the scope from 5 to 25 servers reduces the fraction of lost objects by an order of magnitude, independent of r_L.

larger number of servers. Thus, more access links are available to create additional replicas while the failure lasts. Unless these links are already fully utilized, this increases the average number of replicas per object, and thus improves durability.

Figure 4.14 shows how scope (and thus repair time) affects object durability in a simulation on a synthetic trace. To reduce θ, we limit the bandwidth per server to 1000 B/s in this experiment. We vary the repair threshold and the scope, and measure durability after two years of simulated time. Increasing the scope from 5 to 25 servers reduces the fraction of lost objects by an order of magnitude, independent of r_L. By including more servers (and thus more network connections) in each repair effort, the work is spread over more access links and completes faster, limiting the window of time in which the system is vulnerable to another disk failure. Ideally, by doubling the scope, the window of vulnerability can be cut in half.

A large scope reduces repair time and increases durability; however, implementing a large scope presents two trade-offs. First, the system must monitor each server in the scope to determine the replication levels; when using a large scope, the system must monitor many servers. This increased monitoring traffic limits scalability. Second, in some instances, a large scope can increase the likelihood that a simultaneous failure of multiple disks will cause some object to be lost.

If object replicas are placed randomly with scope N, there are many more objects than disks, and each object has exactly r_L replicas, then it is likely that all $\binom{N}{r_L}$ potential replica sets are used. In this scenario, the simultaneous failure of any r_L disks is likely to cause data loss: there is likely to be at least one object replicated to exactly those disks. A small scope inherently limits

placement possibilities that are used, concentrating objects into common replica sets. As a result, it is less likely that a given set of r_L failures will affect a replica set, but when data loss does occur, many more objects will be lost. These effects are similar: the expected number of objects lost during a large failure event is identical for both strategies. It is the variance that differs between the two strategies.

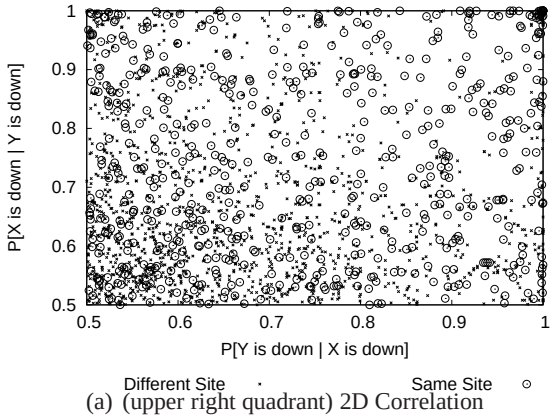
4.3.2 Placement Strategies, Failure Predictors, and Durability

Scope limits the possible servers eligible for replica placement of a particular object; however, once scope has been decided, a plethora of placement strategies are possible. There are two categories of replica placement strategies: random and selective. Random placement is used for its simplicity. Random does not use any information when selecting a server for replica placement. Its use is often accompanied by the assumption that each server failure is independent or has low correlation with each other. If server failures are not independent or have high correlation, the end result could reduce durability or increase costs such as the number of repairs triggered. In contrast, selective placement uses information to choose specific servers that satisfy some constraints. For example, select servers that have been previously shown to have low correlation [AHK+02, DW01, DF82, jBH+05, WMK02]. Selective placement strategies emulate use of a failure predictor that attempts to select servers with the most remaining lifetime and not temporally correlated (fail close together in time).

In this subsection, we explore various random and selective replica placement strategies in the context of systems with PlanetLab failure characteristics. We begin with an analysis of the PlanetLab trace for correlated failures. We use the first interval since it likely contains many correlated failures. Next we describe some selective placement strategies based on analysis. Additionally, we describe some random variants. Finally, we compare replica placement strategies with a simulation using the PlanetLab trace. This study is not exhaustive, however, it demonstrates that even in environments with correlated failures such as PlanetLab, simple placement strategies are sufficient to maintain durability. For example, random replica placement that avoids blacklisted servers and replaces duplicate sites can avoid the problems introduced by the many observed correlated failures.

Correlated Failures in PlanetLab

In this subsection, we test the first interval of the PlanetLab data (Figure 2.3) for the possibility of servers with correlated failures. We test for temporally correlated failures between



(a) (upper right quadrant) 2D Correlation

Site	2D Correlation Threshold	Fraction Correlated
Same	0.5	22.43%
Same	0.8	8.86%
Different	0.5	0.91%
Different	0.8	0.04%

(b) Fraction of Correlated Servers

Figure 4.15: Temporally Correlated Failures. We use a two-dimensional space of conditional downtime probabilities, both p(x is down | y is down) and p(y is down | x is down). Servers x and y are temporally correlated if *both* probabilities are greater than a threshold such as 0.5 or 0.8. (a) upper right quadrant of 2D Correlation. (b) Fraction of Correlated Servers. 22% of the time that a server goes down, there is at least a 50% chance another server in the same site will go down as well. Alternatively, the servers in different sites were not temporally correlated.

pairs of servers that commonly fail close together in time. Servers x and y are temporally correlated if, given server x is unavailable, server y is also likely to be unavailable and visa versa. We perform two tests to measure the degree of temporal correlation. First, we measure the two-dimensional space of conditional downtime probabilities, which illustrates the likelihood that two servers are down at the same time. Using two dimensions reduces the influence of servers with long downtimes. Second, we perform the same study again, however, this time removing the servers with the longest total downtimes. This study illustrates two effects. First the PlanetLab distribution has a long tail of servers with long total downtimes. Second, servers experience correlated failures, however, the strongest correlation is amongst servers in the same site.

First, to capture server correlation, we use a two-dimensional space of conditional downtime probabilities, both p(x is down | y is down) and p(y is down | x is down). Servers x and y are temporally correlated if *both* probabilities are greater than a threshold such as 0.5 or 0.8. Note that most studies only use a single dimension when producing a correlation metric. For instance, prob-

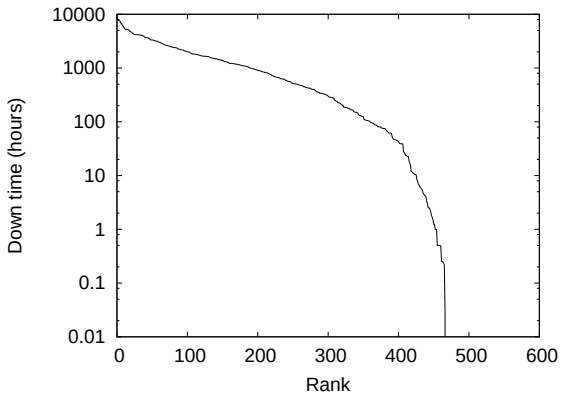
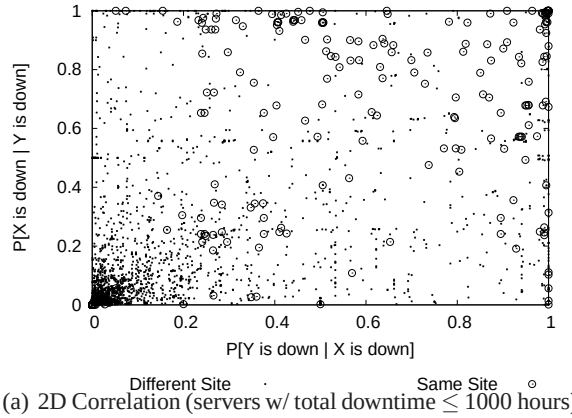


Figure 4.16: Per Server Total Downtime (log scale).

ability that *y* is down given that *x* is down *or* probability that *x* is down given that *y* is down. The use of an "*or*" instead of an "*and*" between the two dimensions increases the number of servers that may be temporally correlated. For example, using only one dimension would correlate servers that are chronically down with all other servers increasing the number of temporally correlated servers; using two dimensions prevents this effect, servers are only temporally correlated if both dimensions are above a threshold. We do not discuss one-dimension conditional downtime probabilities further.

Figure 4.15.(a) shows the upper right quadrant of the two-dimension conditional downtime probability. It highlights servers in the *same site* with open circles and different sites with dots. The fraction of correlated servers are highlighted in Figure 4.15.(b). Figure 4.15.(b) shows that 22% of the time that a server goes down, there is at least a 50% chance another server in the same site will go down as well. Alternatively, Figure 4.15.(b) shows that the two-dimension conditional downtime probability of servers in different sites was insignificant. Figure 4.15 is interesting because it illustrates that significant temporally correlated failures exists.

Second, we look at the two-dimension conditional downtime probability when servers with long downtimes are removed from consideration. Figure 4.16 shows the total downtime for each server and orders the servers from most to least total downtime. Of the 512 total PlanetLab servers, 188 servers have total downtimes greater than 1000 hours. In Figure 4.17.(a), we again show the two-dimension conditional downtime probability, but with servers with total downtime longer than 1000 hours factored out. Figure 4.17.(a) shows more density along the diagonal, meaning that p(*x* is down | *y* is down) is more symmetric with p(*y* is down | *x* is down); that is, the servers are not asymmetrically influenced by long downtimes. Similar to Figure 4.15.(b), Figure 4.17.(b) shows that 33% of the time that a server goes down, who's total downtime is less than 1000 hours, there



(a) 2D Correlation (servers w/ total downtime $\leq$ 1000 hours)

Site	2D Correlation Threshold	Fraction Correlated
Same	0.5	33.33%
Same	0.8	13.83%
Different	0.5	0.34%
Different	0.8	0.04%

(b) Fraction of Correlated Servers (servers w/ total downtime $\leq$ 1000 hours)

Figure 4.17: Temporally Correlated Failures with servers with total downtimes longer then 1000 hours removed from consideration. (a) 2D Correlation and (b) Fraction of Correlated Servers (servers w/ total downtime $\leq$ 1000 hours). 33% of the time that a server goes down, who's total downtime is less than 1000 hours, there is at least a 50% chance another server in the same site will go down. The temporally correlated probability increased when we removed the long downtime servers because the number of servers temporally correlated remained relatively unchanged from Figure 4.15 while the total number of servers was reduced by 188. Alternatively, the servers in different sites were not temporally correlated.

is at least a 50% chance another server in the same site will go down. The temporally correlated probability increased when we removed the long downtime servers because the number of servers temporally correlated remained relatively unchanged while the total number of servers was reduced by 188. However, the inter-site two-dimension conditional downtime probability is still insignificant. Figures 4.16 and 4.17 are interesting because they demonstrates that significant correlation exists amongst servers in the same site and not much between servers in different sites.

Replica Placement Strategies

Unreliable and correlated servers have been cited in literature [CV03, NYGS04, YNY+04]. However, it is not clear to what degree "flaky" and correlated servers affect durability or cost such as the number of repairs triggered. We compare random placement strategies that blacklist flaky servers and/or avoid placing multiple replicas in duplicate sites. In particular, we blacklist the top

10% of servers that are likely to be unavailable; 22% of PlanetLab servers in the first interval are available less than 50% of the time according to Figure 2.3.(d)). Additionally, to avoid placing multiple replicas in duplicate sites, we pick another random server to store a replica if a server in a duplicate site was already selected.

The four variations of random replica placement strategies that we compare are `Random`, `RandomBlacklist`, `RandomSite`, and `RandomSiteBlacklist`. `Random` placement picks *n* unique servers at random to store replicas. `RandomBlacklist` placement is the same as `Random` but avoids the use of servers that show long downtimes. The blacklist is comprised of the top *z* servers with the longest total downtimes. `RandomSite` avoids placing multiple replicas in the same site. `RandomSite` picks *n* unique servers at random and avoids using servers in the same site. We identify a site by the 2B IP address prefix. The other criteria can be geography or administrative domains. Finally, `RandomSiteBlacklist` placement is the combination of `RandomSite` and `RandomBlacklist`.

The other category of replica placement is selective. The benefits of more sophisticated selective placement strategies are not well understood in terms of durability and costs. We compare costs of the random placement strategies discussed above against the best selective placement strategy that uses a failure predictor. Our failure predictor uses future knowledge when selecting servers for replica placement. Future knowledge is based on offline information (i.e. the PlanetLab trace).

This offline clairvoyant selective replica placement strategy, named `Max-Sum-Session`, uses future knowledge of server lifetimes, sessiontimes, and availability to place replicas. In particular `Max-Sum-Session` places replicas on servers with the highest remaining sum of sessiontimes. This strategy places replicas on servers that permanently fail furthest in the future and exhibit the highest availability. The `Max-Sum-Session` was the best performing algorithm of all clairvoyant algorithms we studied (e.g. `Max-Sum-Session` performed better than some anti-correlation techniques [WMK02]).

Evaluation of Replica Placement Strategies

We now compare different replica placement strategies. We compare different random, DHT, and clairvoyant replica placement strategies. DHT is `Random` but with a small scope; all other strategies have a large scope of size *N* (all servers in system). Table 4.1 shows for all the placement strategies the total number of repairs triggered and percentage of improvement over `Random`. Additionally, Table 4.1 shows the average and standard deviation of the number of replicas per

	Data Replica Strategy. ($r_L = 5$, $n = 11$, and $\|\text{blacklist}\| = 35$)					
	Random	DHT	RandomSite	RandomBlacklist	RandomSiteBlacklist	Max-Sum-Session
# Repairs	227,242	447,204	223,003	221,618	217,291	209,815
% Improvement		-96.80	1.87	2.47	4.38	7.67
# Replicas/N	1386.2	1439.9	1381.9	1391.8	1386.5	1388.9
Stddev ($\pm$)	1126.6	684.4	1140.5	1182.1	1193.7	1213.3

Table 4.1: Comparison of Replica Placement Strategies. $r_L = 5$ and $n = 11$.

server. The storage system used a low watermark of $r_L = 5$. The size of the blacklist for the RandomBlackList and RandomSiteBlacklist placement strategies was the top 35 servers with the longest total downtimes. Table 4.1 shows how the replica placement strategies differ in cost (number of repairs triggered).

Table 4.1 shows that more sophisticated placement strategies exhibited noticeable increase in performance; that is, fewer repairs triggered compared to Random. For example, the RandomSiteBlacklist placement showed a 4.38% improvement over Random, which was slightly more than the sum of parts, 1.87% and 2.47% for RandomSite and RandomBlacklist, respectively. The clairvoyant placement strategy exhibited a 7.67% improvement (Max-Sum-Session). The DHT placement triggered more data repairs than Random. However, because the load balance is a primary goal of consistent hashing used by DHT, the distribution of the number of replicas per server was more uniform for DHT as can be seen with the smaller standard deviation of 684 replicas per server.

4.4 Summary

Fault tolerance, the first key to ensuring durability, is a property that is highly desired in distributed wide-area storage systems, yet setting values for its parameters are often not well understood. A fault tolerance algorithm must choose the type of redundancy, the number of replicas to create, and where to store replicas. However, questions arise such as what redundancy scheme should be used? How much redundancy is needed to tolerate failure? What is the associated durability? Where to store replicas? What servers should and should not be eligible to store replicas. These are all questions that govern a systems ability to tolerate failure without loss of data. In this chapter, we explored many techniques to answer these questions using combinations of analytical models and simulation. We discuss four insights and techniques to answer these questions.

First, we quantitatively compared systems based on replication to systems based on erasure-codes. We showed that the mean time to data loss (MTTDL) of an erasure encoded system is often

many orders of magnitude higher than that of a replicated system with the same storage overhead and repair period. Another interesting result of our analysis showed that erasure-resilient codes use an order of magnitude less bandwidth and storage than replication for systems with similar MTTDL. Use of erasure-codes, however, has negative consequences. Erasure-codes increase system complexity. Complexities include increased memory and processor utilization to produce fragments, use of cryptographic mechanisms to identify erasures and ensuring the integrity of fragments and data, need for aggregation, and reconstructing data from fragments for repair. Ultimately, as a result of these complexities, a designer must decide if the efficiency of erasure codes is more valuable than the simplicity of replication.

Second, we presented a model that helps the system select the number of replicas to create for an object. The proper number of replicas to create is related to the burstiness of permanent failures. We use a model that computes the durability (probability that a data object exists after a specific amount of time) given a rate of server failure (replica loss), rate of replica creation, and target replication level. The model based on a continuous time Markov chain calculates the probability that a burst of server failure destroys the remaining replicas before more replicas can be created in a specific amount of time. Replica creation rate is dependent on replica placement, repair scheduling, and constrained by resources such as access link bandwidth that restrict the number of objects that a server can repair in a given time period. This model considers repair scheduling and constrained resources; placement, however, is considered separately. This model allows the system to calculate a target replication level that satisfies a target durability constraint.

Third, we demonstrated that durability could be increased by copying object replicas from a failed server in parallel. In effect, reducing the time needed to repair the replicas lost on the failed servers. The decrease in repair time increases durability since durability is inversely proportional to repair time [PGK88]. We call this property scope. Scope determines the number of servers that are eligible to store replicas for a particular object. Also, it limits which particular servers are eligible. Furthermore, after replicas are placed, scope implies which servers are monitored for liveness and storage of replicas for particular objects. Increasing scope increases parallelism. However, increasing scope also requires the system to monitor more servers limiting scalability. Section 5.2.1 illustrates that monitoring costs are often insignificant when compared to repair costs.

Finally, deciding where to place replicas is difficult since there are many possibilities. However, in the context of durability, two properties are most relevant: scope (discussed above) and predictability. Ideally, if the system knew the exact time that servers fail, then it could select servers to store replicas based on two criteria. First, servers that fail furthest in the future. However,

this technique results in an unbalanced system since some servers store significantly more data than others. Second, select servers where failures are spaced far apart in time. This technique avoids temporally correlated failures (failures that occur close together in time) allowing repair more time to replace lost replicas before another failure occurs. The problem, of course, is that an oracle that knows the exact failure time is not available to systems. However, we demonstrated that a random placement policy—that avoids blacklisted servers and duplicate sites—is sufficient to avoid many observed correlated failures. For instance, such a policy triggered only 3.4% more repairs than a clairvoyant placement that knew the future time that servers fail.

Chapter 5

Repair

Repair is second key to ensuring data durability. The goal of repair is to refresh lost redundancy before data is lost due to permanent failures. Monitored information, which measures the number of available replicas, is the basis for initiating repair. However, this monitored information is imprecise since replicas can be durably stored but not immediately available, hence transient failure. The possibility of transient failures complicates providing durability efficiently: we do not want to make new copies in response to transient failures, but it is impossible to distinguish between disk failures and transient failures using only remote network measurements. This chapter focuses on minimizing the amount of network traffic sent in response to transient failures while maintaining a target durability. We demonstrate three techniques to reduce cost due to transient failure.

First, in Section 5.1, we show that timeouts reduce costs due to transient failure; however, their effectiveness is limited. Timeouts reduce false-positives, misclassifying servers as permanently failed that have actually only transiently failed. Their effectiveness is dependent on the downtime distribution. If the timeout value is set to mask most of the downtime distribution, a transiently failed server may return before a timeout expires and prevent resources from being wasted creating replicas unnecessarily. However, setting longer timeout values decrease durability. The time to recognize permanently failed servers increases as the timeout value increases, thus increasing the "window of vulnerability" for an object to be lost due to permanent failure.

Second, in Section 5.2, we show that extra replicas, beyond what is required to maintain a target durability, decreases cost due to transient failure. We demonstrate that the number of extra replicas that minimizes cost can be estimated. The advantage of extra replication is that it exponentially reduces the cost due to transient failure with only a linear increase in replicas. Furthermore, there is no reduction in durability by adding extra replicas. As a result, extra replicas perform better

than long timeouts.

Finally, in Section 5.3, we show that the optimum number of extra replicas does not need to be estimated at all. Costs are minimized by simply responding to all failures, transient or permanent, creating replicas until r_L (number of replicas required for target durability) are available, tracking all replicas created, and reintegrating replicas after transient failure. We call such an algorithm Carbonite. Carbonite was first presented in [CDH+06], we provide a description here for completeness. The result of the Carbonite algorithm is that the system performs some extra work for each object early in its life, but over the long term creates new copies of the object only as fast as it suffers permanent failures.

We assume that all data is immutable in Sections 5.1, 5.2, and 5.3. We consider efficiently maintaining durability of mutable data in Part III.

5.1 Reducing Transient Costs with Monitoring and Timeout-based Failure Detectors

Most replication systems are closed-loop systems; they sense, and can respond to, the current state of the system. For example, many systems monitor server failures and keep track of the set of replicas that is stored on each server. Once a failure is detected, these systems can respond by creating a new replica of each object that was stored on the failed servers [BTC+04, CDH+06, Cat03, DKK+01, HMD05, RGK+05]. In particular, we assume that durability is maintained by monitoring the availability of at least a low watermark of r_L replicas for a particular object (Section 4.2). When the number of available replicas drops below r_L, more replicas are created until one of two situations occur: either r_L replicas are again available or no replicas are available, the object may have been lost (see Section 4.2 for a detailed analysis of r_L and durability).

In this section, we describe failure detectors and how their accuracy affects durability and cost to maintain durability. Cost is measured as the number of replicas created (i.e. total bytes sent) to maintain durability. In particular, we demonstrate that longer timeout values increase the accuracy of timeout-based failure detectors and decrease the cost to maintain durability; however, longer timeouts decrease durability since more time is required to recognize permanent failures.

We do not discuss, in this section, the cost due to monitoring and how to build a monitoring infrastructure, we leave those discussions for Sections 5.2 and 9.2, respectively.

5.1.1 Failure detectors

A very common information source for replication systems is the *failure detector*. A failure detector is a function $f(N,t) \rightarrow \{A,U,D\}$ which, for a given time t, maps every server N to one of three states: Available, unavailable, or dead. The second state models transient failures, during which the server cannot be reached by the other servers[1] but otherwise remains intact, while the third state models permanent failures with data loss, such as hardware failures. The information from the failure detector can be used e.g. to drive the creation of new replicas.

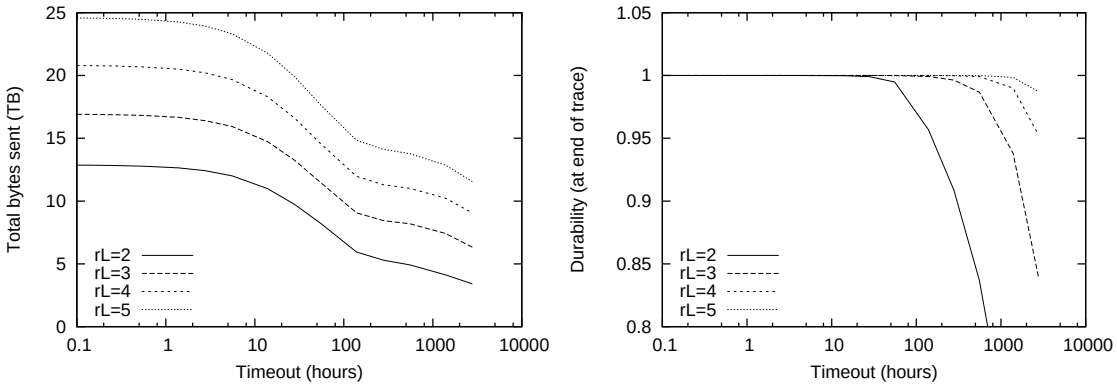
A perfect failure detector is *complete* and *accurate*, meaning that it detects all failures and does not report a failure unless one has actually happened. Ideally, it is also *instantaneous*, meaning that failures are reported immediately. Unfortunately, real failure detectors do not have all of these properties. An incomplete failure detector may not report a failure at all, and one with delays may do so too late for the system to respond. Both effects can cause data loss, which the system can avoid by creating additional redundancy. Inaccuracies do not cause data loss but may cause extra overhead, since they may prompt the system to create unnecessary replicas.

In distributed systems, transient failures are usually detected by sending ping messages to a remote host, and by declaring it unavailable when no answer is received within a short timeout τ_1. Since permanent failures have the same symptoms, the assumption is usually made that transient failures do not last longer than a maximum time τ_2; the server is declared dead when it has not responded to pings for at least that time. Measurement studies of existing systems [BDET00, YNY+04] have shown that this assumption is realistic.

Timeout-based failure detectors are complete (a failed server cannot respond to pings and therefore *will* be declared dead after τ_2), but they are not completely accurate, and they have an inherent delay. There is a difficult tradeoff between delay and accuracy. By increasing the timeouts, we can reduce the false positives; however, this comes at the cost of a higher delay, which increases the probability that multiple failures occur before the system can detect the first one.

Better failure detectors can be built if special hardware is available, e.g. a watchdog [Fet03]. However, in today's wide-area storage systems, such hardware is generally not available.

[1] A server is not reachable by other servers possibly due to temporary server failure such as reboot or network failure such as dropped messages or network partion.



(a) (b)

Figure 5.1: The impact of timeouts on bandwidth and durability on a synthetic trace. Figure 5.1(a) shows the number of copies created for various timeout values; (b) shows the corresponding object durability. In this trace, the expected downtime is about 29 hours. Longer timeouts allow the system to mask more transient failures and thus reduce maintenance cost; however, they also reduce durability.

5.1.2 Evaluation of Timeout-based Failure Detectors

Timeout-based failure detectors are most effective when a significant percentage of the transient failures can be ignored, which is dependent on the downtime distribution (e.g. Figure 2.5.(c) illustrates the downtime distribution for PlanetLab). However, for durability to remain high, the expected server lifetime needs to be significantly greater than the timeout.

To evaluate this scenario where timeouts should have impact, we performed an experiment using a synthetic trace where we varied the repair threshold r_L and the server timeout[2]. Since the system would recognize servers returning after a permanent failure and immediately expire all pending timeouts for these servers, we assigned new identities to such servers to allow long timeouts to expire normally.

Figure 5.1 shows the results of this simulation: (a) shows the total bytes sent as a function of timeout while (b) shows the durability at the end of the trace. As the length of the timeout increases past the average downtime, we observe a reduction in the number of bytes sent without a decrease in durability. However, as the timeout grows longer, durability begins to fall: the long

[2]In the simulaor, the server timeout is a system-wide defined parameter. An adaptive scheme such as setting a timeout value per server may perform better [LSMK05] (e.g. using a distribution of downtime per server such as Figure 4.16 may reduce the total bytes sent by masking transient failures). However, use of timeouts (either statically or dynamically) may reduce durability since a timeout inherently increases the lag time to detect a permanent failure, thus increasing the 'window of vulnerability', during which additional failures can cause data loss.

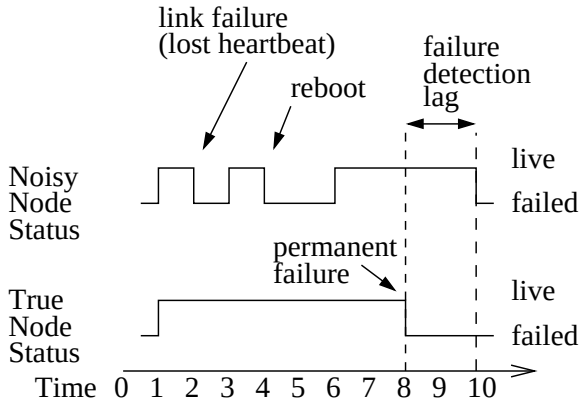


Figure 5.2: Transient and Permanent Failures over Time.

timeout delays the point at which the system can begin repair, reducing the effective repair rate. Thus setting a timeout can reduce response to transient failures but its success depends greatly on its relationship to the downtime distribution and can in some instances reduce durability as well.

5.2 Reducing Transient Costs with Extra Replication

Storage systems are required to trigger repair and replace lost replicas as servers fail to maintain target data durability levels. As described earlier, a fundamental problem with this construction is differentiating permanent failures (data is lost) from transient failures (server returns with data intact). Figure 5.2 shows an example of transient and permanent failures over time. Transient failures that render a server temporarily unavailable are due to server reboot, system maintenance, Internet path outage, etc. In addition to transient failures, failure detection has a lag; a permanently failed server is classified as alive during the lag period. A study found that transient failures occur often in the wide-area [CV03].

Triggering repair due to transient failures can increase the cost of maintaining data severely. Some environments are very reliable and do not have much transient failures (e.g. within a data center). In contrast, other environments do not support durable storage [BR03] due to too many permanent and transient failures (e.g. Kazaa, Gnutella, and other high client churn environments). But for many wide-area systems, like PlanetLab [BBC$^+$04], reliable storage can be supported and transient failures are common[CV03].

The cost of maintaining durability is determined by the cost due to permanent failure, transient failure, write, and monitoring rate.

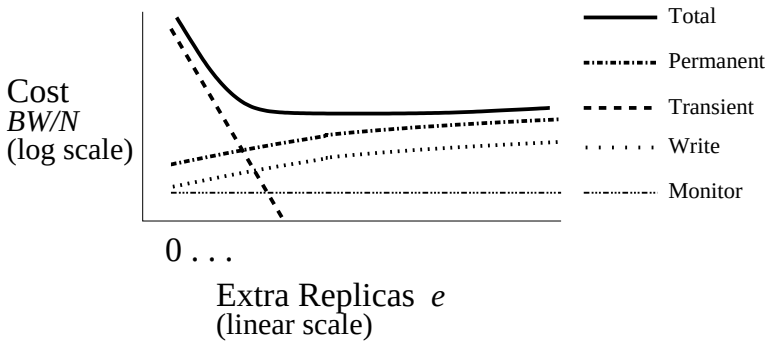


Figure 5.3: Example. Cost per Server of Maintaining Data in a Wide Area Storage System.

$$\begin{aligned} \text{Work} &= f(\text{permanent failure}, \text{transient failure}, \text{write}, \text{monitoring}) \\ &= f(\text{permanent failure}) + f(\text{transient failure}) + \\ &\quad f(\text{write}) + f(\text{monitoring}) \end{aligned} \tag{5.1}$$

Although the overall cost is additive, the cost due to transient failures often dominates. Intuitively, decreasing the cost due to transient failures decreases the overall cost.

In this section, we show that we can decrease the cost due to transient failures by adding extra replicas beyond what is required for a target durability. Decreasing the reaction to transient failure is analogous to decreasing the error rate of sending a message across a noisy channel by adding extra bits. Explicitly, we tradeoff increased storage for reduced communication while maintaining the same minimum[3] target durability. The extra replicas absorb noise. This translates into a decreased rate of triggering repair since it is less likely for the extra replicas to *simultaneously* be unavailable. Figure 5.3 illustrates the breakdown in the data maintenance costs as the extra replicas are increased.

Figure 5.3 is key. It shows that with no extra replicas the probability of triggering repair due to transient failures is actually quite high; hence, the cost due to transient failures is high. In fact, the probability of triggering repair is *higher* than the probability of a single timeout since the chance of *any* one out of n replicas timing out is higher than the probability of a single timeout. If we add extra replicas and require that at least *all* extra replicas simultaneously be unavailable, then

[3]Extra replication beyond what is required for durability does increase the expected durability since more replicas must be lost for the object to be lost. However, we use it to decrease communication costs and hence call the target durability, the minimum target durability.

the rate of triggering repair drops significantly; similarly, the cost due to transient failures drops significantly.

The goal is to estimate the optimum number of extra replicas required to minimize work. In the following subsections we describe how to estimate this optimum number of extra replicas. In particular, we discuss how to estimate the number of extra replicas that minimize work.

Note, this estimator algorithm assumes that any replicas that exist, but were unavailable when repair was triggered are *forgotten* about; when the extant replica returns it is *not* re-included into the replica set.

5.2.1 Estimator Algorithm

In this section, we show how to estimate the amount of extra replicas required to absorb "noise" and reduce rate of triggering repair. The algorithm works as follows. Given a target data durability to maintain, first it calculates the minimum low watermark r_L of the number of replicas required to be available (see Section 4.2 for relationship between r_L and durability, calculation based on ratio between the rate of replica creation μ [access link speeds, replicas per server, and replica placement] and rate of server failure λ_f, i.e. r_L is based on $\mu/\lambda_f = \theta$). It then supplements this number with a set of extra replicas to absorb noise (calculation based on average server lifetime, sessiontime, and downtime). Finally, it triggers repair when *all* extra replicas are simultaneously considered unavailable. In the following subsections we show how to estimate the amount of extra replicas. See Section 4.2 to understand how to first set r_L.

Estimating the Amount of Extra Replicas

We estimate the optimum number of extra replicas e by synthesizing the cost due to maintaining durability as expressed in Equation 5.1. In particular, we develop an estimator for each term in Equation 5.1, then calculate each term's cost, and pick e where the overall cost is minimum. The key is to pick the optimum number of extra replicas e that reduces the cost due to transient failures without increasing the cost due to writes or permanent failures too high. We discuss each term's estimator and the overall extra replication estimator in turn below.

Permanent Failure Estimator

Permanent failure is the loss of data on a server. The maintenance cost due to permanent failures is dependent on the average amount of storage per server $\frac{S}{N}$ and the average storage server lifetime T.

Variable	Description
r_L	Number of replicas required to be available. Repair is invoked when available replicas is less than r_L.
e	Number of "extra" replicas beyond r_L to create during repair.
n	Total number of replicas available after repair completes $(n = r_L + e)$.
m	Number of replicas required to read object. $m = 1$ replication, $m > 1$ erasure codes.
$1-\varepsilon$	Target minimum data availability.
ε	Probability data is *un*available.
a	Average server availability.
D	Total amount of unique data.
S	Total amount of storage ($S = kD$) where storage overhead factor $k = \frac{n}{m}$.
N	Total number of servers.
T	The average lifetime of all N servers.
$u(x)$	Probability distribution function of downtimes
τ	Timeout used to determine a server is unavailable.
τ_{max}	Maximum time a server has been unavailable and came back.
p_{dr}	Rate of triggering repair
p_τ	Probability server down longer than timeout τ.

Table 5.1: Notation

$$\text{permanent}\frac{BW}{N} \quad = \quad O\left(\frac{S}{NT}\right) \tag{5.2}$$

where total storage S is the product of the total amount of unique data D and storage overhead factor k (i.e. $S = kD$). Note that the storage overhead factor is dependent on the rate of erasure-coding $\frac{n}{m}$ ($m = 1$ for replication) and total number of replicas n; that is, $k = \frac{r_L+e}{m} = \frac{n}{m}$. Equation 5.2 states that on average the total storage S must be copied to new storage servers every average server lifetime T period. We assume that all storage servers have a finite lifetime (e.g. 1-3 years) typical of a commodity server, so storage will not be biased towards one ultra reliable server. Equation 5.2 has been discussed in literature by Blake and Rodrigues [BR03].

Transient Failure Estimator

Transient failure is when a server returns from failure with data intact. Reducing the rate of triggering repair due to transient failure reduces the amount of unnecessary repair. We assume that a timeout-based failure detector with value timeout τ is used to determine if a server has failed or not. p_τ is the probability of a single timeout. If there are no extra replicas, then the probability of *at least* one server timing out is high; as a result, the rate of triggering data repair, p_{dr}, is high. That

is,

$$\begin{aligned} p_\tau &= P(\text{storage unavailable longer than } \tau) \\ &= \int_\tau^\infty u(x)dx \qquad (5.3) \\ p_{dr} &= P(\text{at least one storage server unavailable longer than } \tau) \\ &= \sum_{i=1}^{n=r_L} \binom{n}{i} p_\tau^i (1-p_\tau)^{n-i} \qquad (5.4) \end{aligned}$$

where $u(x)$ is the probability distribution function of downtimes. Figure 2.3(c) is the downtime distribution for PlanetLab. Equation 5.4 is the probability that at least one of the n replicas times out. Note that the probability of at least one of n replicas timing out is higher than the probability of a single time out p_τ. This assumes that failures are independent.

Lets now assume we add extra replicas beyond r_L. We require that at least *all* extra replicas to simultaneously be unavailable in order to trigger repair. As a result, the rate of triggering repair is

$$\begin{aligned} p_{dr} &= P(\text{at least all } \textit{extra} \text{ replicas unavailable longer than } \tau) \\ &= \sum_{i=e+1}^{n=r_L+e} \binom{n}{i} p_\tau^i (1-p_\tau)^{n-i} \qquad (5.5) \end{aligned}$$

Equation 5.5 computes the probability that at least $e+1$ servers have simultaneously timed out. It also shows that the rate of triggering repair can be reduced by increasing the extra replicas. For example, given a timeout period $\tau = 1$ hour and a probability of a timeout $p_\tau = 0.25$, then for the following parameterization $m = 1, r_L = 5, e = 4 (n = 9 = 5+4)$, the resulting rate of triggering repair is $p_{dr} = 0.049$, which is significantly less than the probability of triggering repair with no extra replicas $p_{dr} = 0.762$ ($m = 1, r_L = 5, n = 5, e = 0$).

The cost of triggering repair is the amount of storage per server $\frac{S}{N}$ and the average period for the MTTF and MTTR (i.e. average session and downtime). The transient term is

$$\text{transient}\frac{BW}{N} = p_{dr} \cdot O\left(\frac{S}{N(MTTF+MTTR)}\right) \qquad (5.6)$$

Write Rate Estimator The write rate is the rate of unique data being added to the storage system. The cost due to writes is simply the unique write rate multiplied by the storage overhead factor k.

$$\text{write}\frac{BW}{N} \quad = \quad k \cdot \text{write rate} \tag{5.7}$$

Ideally, we want $k = \frac{r_L+e}{m} = \frac{n}{m}$ to be small. However, the cost due to writes may be increased until an optimum number of extra replicas e is obtained since k depends on e.

Heartbeat Timeout-based Failure Detector Estimator

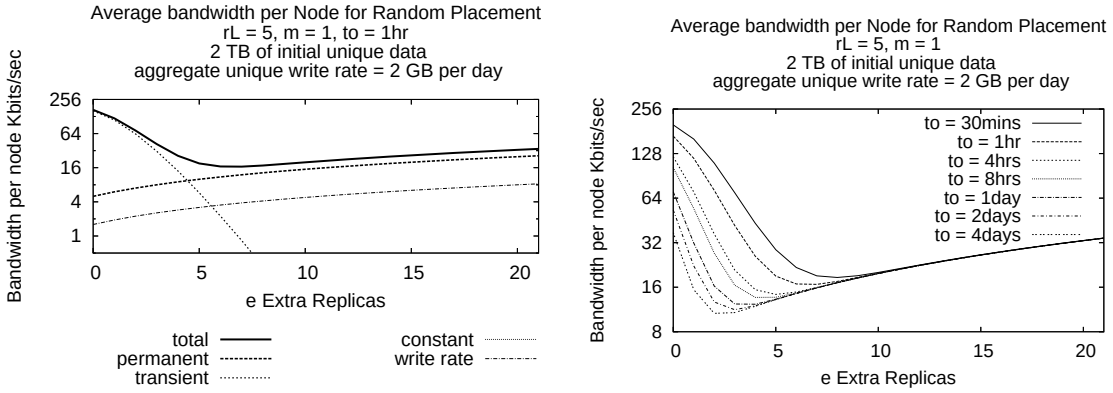
A heartbeat timeout-based failure detector is used to determine whether a server is alive or not. We assume that each server heartbeats all other servers, as a result, knows the status of all other servers. This assumption is used to implement a monitoring infrastructure called a distributed directory (see Sections 9.2). The cost per server for monitoring all other servers is dependent on the number of servers N, the heartbeat timeout period τ, and the size of a heartbeat hb_{sz}, That is,

$$\text{heartbeat}\frac{BW}{N} \quad = \quad \frac{N}{\tau} \cdot hb_{sz} \tag{5.8}$$

Equation 5.8 states that each server sends a heartbeat to all other servers every τ period. For many reasonable timeouts, the cost due to heartbeats will not be a significant fraction of the overall data maintenance costs. For example, if $N = 10,000$ servers, $\tau = 1$ hour, and $hb_{sz} = 100$ B, then $\text{heartbeat}\frac{BW}{N} = 277.8$ Bps.

Example of Applying Extra Replication Estimator

Figure 5.4 shows an example of applying the extra replication estimator. Using an $r_L = 5$, we can maintain six 9's of data durability (i.e. 1 out of every million objects is permanently lost per year). We assume network access link speeds of 1.2Mbps (or 150KB/s), an aggregate amount of unique data is $D = 2TB$, the aggregate unique write rate is $2GB$ per day, the number of servers is $N = 400$, and the timeout value is $\tau = 1$ hour. Finally, we use the expected server availability, lifetime, MTTF, and MTTR from Figure 2.3(d) and the downtime distribution from Figure 2.3(c). Using an average of 400 servers in the system, we can estimate the mean time between failures for a single disk as $400 \cdot 39.85$ hours or 1.82 years. Hence, $\lambda_f \approx 0.550$ disk failures per year. The replica creation rate $\mu \approx 187$ disk copies per year given 25GB of replicated data per server and 150KB/s



(a) $\tau = 1$ hour (b) Vary τ

Figure 5.4: Extra Replication Estimator for Storage Systems on PlanetLab.

per server network access link speeds. Thus, $\theta \approx 320$ and $r_L = 5$ yields six 9's of durability.

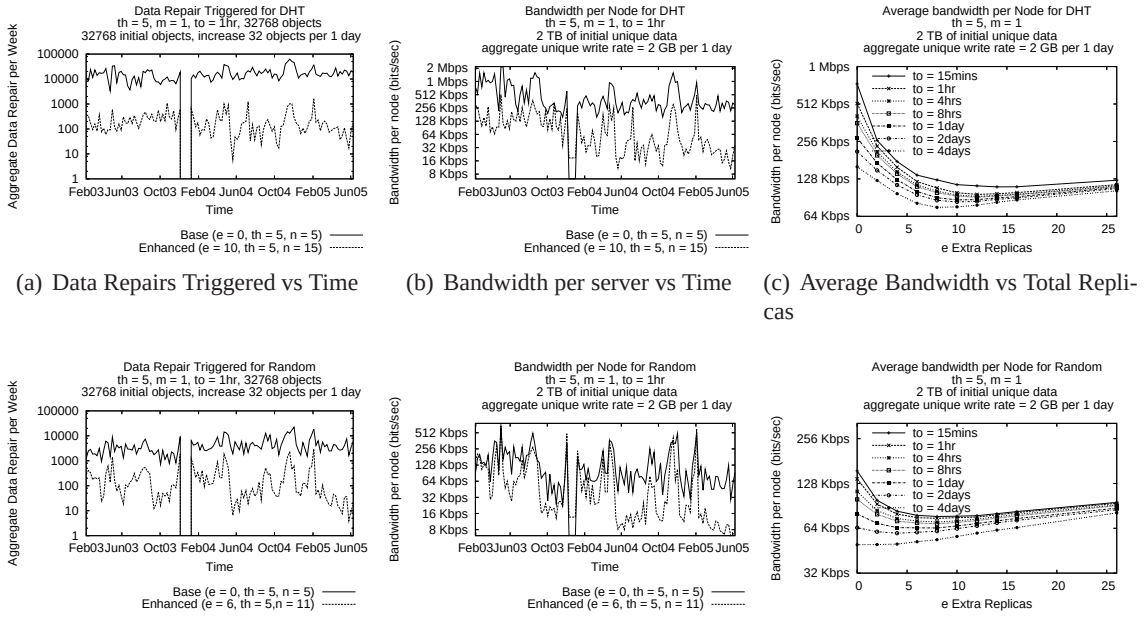
Figure 5.4(a) shows that the estimated optimum number of extra replicas that minimizes the cost due to data maintenance is six. Similarly, Figure 5.4(b) shows the estimated optimum number of extra replicas for varying timeout values.

Notice that the extra replication estimator computation can be performed locally at a each server with local estimates for the total number of storage servers, average server availability, lifetime, MTTF, MTTR, storage per server, and write rate per server. This is beneficial because the extra replication estimator can be performed online so that the storage system parameterization can adapt to the changing environment characteristics overtime.

5.2.2 Evaluation of Extra Replication

We now evaluate the effectiveness of extra replication. For this analysis, we maintain six 9's of data durability, use a low watermark of $r_L = 5$, limit the access link speeds to 1.2Mbps (150KB/s), and use `Random` placement with large and small scope denoted by `Random` and `DHT` respectively. We used a timeout value of $\tau = 1$hr. In Figure 5.5, we measure the number of repairs triggered and average bandwidth per server over time for the optimum and worst number of extra replicas. Additionally, we show the breakdown in cost in Figure 5.6. Note that the small scope, DHT-based storage system, parameterization (i.e. $m = 1$, $r_L = 5$, and small scope) is the same as Dhash [Cat03, DKK+01].

The results in Figure 5.5 show that in both the small scope (DHT) and large scope `Random`-based storage system (Figures a-c and d-f, respectively), the configurations that use the estimated



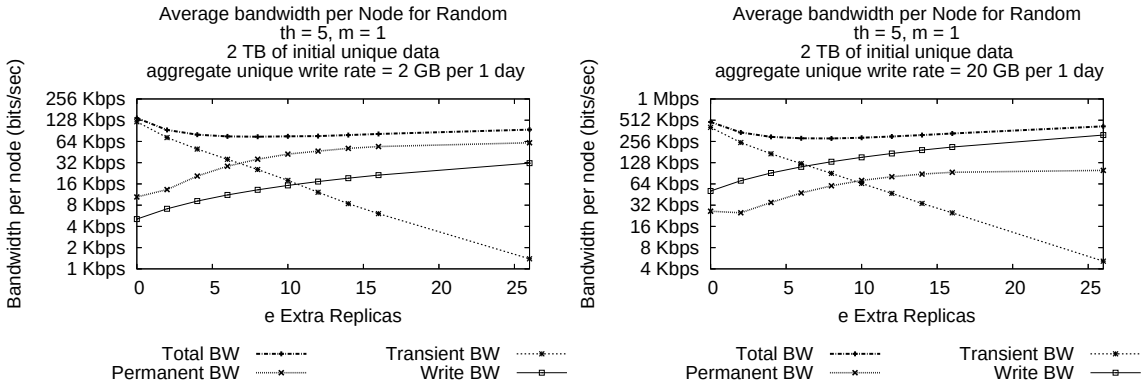
(a) Data Repairs Triggered vs Time (b) Bandwidth per server vs Time (c) Average Bandwidth vs Total Replicas

(d) Data Repairs Triggered vs Time (e) Bandwidth per server vs Time (f) Average Bandwidth vs Total Replicas

Figure 5.5: Extra Replication. Figures (a), (b), and (c) use the DHT-based storage system like Dhash and Figures (d), (e), and (f) use a directory-based storage system with a Random placement. Figures (a) and (d) shows the number of repairs triggered per week over the course of the trace. Figures (b) and (e) show the average bandwidth per server (averaged over a week) over the course of the trace. Finally, Figures (c) and (f) show the average bandwidth per server as we vary the number of extra replicas and timeout values.

optimum number of extra replicas use at least an order of magnitude less bandwidth per server than with no extra replicas. Furthermore, Figures (c) and (f) show that large timeout values exponentially decrease the cost of data maintenance; however, the increase in server failure detection potentially compromises data durability. An alternative solution was a linear increase in extra replicas which similarly exponentially decreased the cost of data maintenance without sacrificing data durability. Figures (c) and (f) are consistent with the expected costs as depicted in Figure 5.4(b).

Figure 5.6 showed the breakdown in bandwidth cost for maintaining a target durability and extra replicas. Figure 5.6 fixed both the timeout $\tau = 1$hr and data placement strategy to `Random` with a large scope. Figure 5.6(a) and (b) used a per server unique write rate of 1Kbps and 10Kbps, respectively. Both Figures 5.6(a) and (b) illustrated that the cost of maintaining data due to transient failures dominated the total cost. The total cost was dominated by unnecessary work. As the number of extra replicas, which are required to be simultaneously unavailable in order to trigger repair, increased, the cost due to transient failures decreased. Thus, the cost due to actual perma-



(a) Cost Breakdown vs Number Replicas (1Kbps) (b) Cost Breakdown vs Number Replicas (10Kbps)

Figure 5.6: Cost Breakdown for Maintaining Minimum Data Availability for 2 TB of unique data. (a) and (b) Cost breakdown with a unique write rate of 1Kbps and 10 Kbps per server, respectively. Both (a) and (b) fix the data placement strategy to `Random` and timeout $\tau = 1$hr. The cost due to heartbeats is not shown since it was less than 1Kbps.

nent failures, which was a system fundamental characteristic, dominated. The difference between Figure 5.6(a) and (b) is that the cost due to permanent failures dominated in (a) and the cost due to new writes dominated in (b). Finally, the cost due to sending heartbeats to each server in an all-pairs ping fashion once an hour was insignificant. These results are consistent with the extra replication estimator depicted in Figure 5.4(a).

5.3 Reducing Transient Costs with Reintegration

In Section 5.2, we demonstrated that extra replication reduces cost due to transient failures. However, the estimator algorithm (Section 5.2.1) involved the measurement of many values which all affect the accuracy of the algorithm and may be hard to measure in practice. In this section, we illustrate a simpler algorithm to add extra replicas to an object where no values need to be estimated.

The key technique needed to achieve this is to ensure that the system reintegrates object replicas stored on servers after transient failures. For reintegration to be effective, we assume that the system is able to track all replicas created for an object and objects are immutable[4]. The number of replicas that the system must remember turns out to be dependent on *a*, the average fraction of time that a server is available. However, we show that the correct number of extra replicas can

[4]If objects are mutable and an update occurrs while a replica is unavailable, then reintegrating the server may actually increase costs since the replica needs to be updated.

```
// Iterate through the object database
// and schedule an object for repair if needed
MAINTAIN_REPLICAS ()
  keys = <DB.object_keys sorted number of available replicas>
  foreach k in keys:
    n = replicas[k].len ()
    if (n < r_L)
      newreplica = enqueue_repair (k)
      replicas[k].append (newreplica)
```

Figure 5.7: Each server maintains a list of objects for which it is responsible and monitors the replication level of each object using some synchronization mechanism. In this code, this state is stored in the replicas hash table though an implementation may choose to store it on disk. This code is called periodically to enqueue repairs on those objects that have too few replicas available; the application can issue these requests at its convenience.

be determined without estimating a by tracking the location of all replicas, including those that are offline. Carbonite is an algorithm that uses this technique. We demonstrate its effectiveness using simulations. We additionally show that reintegration is effective for storage systems that use erasure-coding. Carbonite was first presented in [CDH+06]. We provide a description here for completeness.

5.3.1 Carbonite details

The Carbonite maintenance algorithm focuses on reintegration to avoid responding to transient failures. Durability is provided by selecting a suitable value of r_L; an implementation of Carbonite should place objects to maximize θ and preferentially repair the least replicated object. Within these settings, Carbonite works to efficiently maintain r_L copies, thus providing durability.

Because it is not possible to distinguish between transient and disk failures remotely, Carbonite simply responds to any detected failure by creating a new replica. This approach is shown in Figure 5.7. If fewer than r_L replicas are detected as available, the algorithm creates enough new replicas to return the replication level to r_L.

However, Carbonite remembers which replicas were stored on servers that have failed so that they can be reused if they return. This allows Carbonite to greatly reduce the cost of responding to transient failures. For example, if the system has created two replicas beyond r_L and both fail, no work needs to be done unless a third replica fails before one of the two currently unavailable replicas returns. Once enough extra replicas have been created, it is unlikely that fewer than r_L of

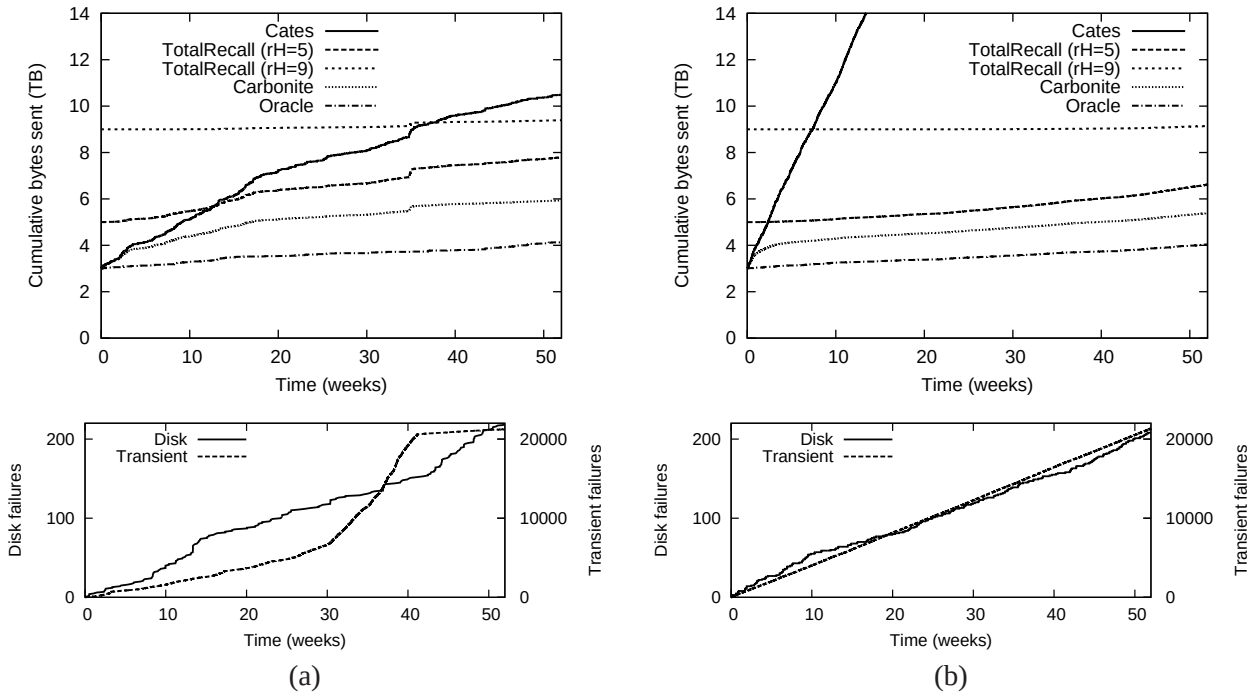


Figure 5.8: A comparison of the total amount of work done by different maintenance algorithms with $r_L = 3$ using a PlanetLab trace (left) and a synthetic trace (right). In all cases, no objects are lost. However, $r_L = 2$ is insufficient: for the PlanetLab trace, even a system that could distinguish permanent from transient failures would lose several objects.

them will be available at any given time. Over time, it is increasingly unlikely that the system will need to make any more replicas.

5.3.2 Reintegration reduces maintenance

Figure 5.8 shows the importance of reintegrating replicas back into the system by comparing the behavior of Carbonite to two prior DHT systems and a hypothetical system that can differentiate disk from transient failures using an oracle and thus only reacts to disk failures. In the simulation, each system operates with $r_L = 3$. The systems are simulated against the PlanetLab trace (a) and a synthetic trace (b). The *y*-axes plot the cumulative number of bytes of network traffic used to create replicas; the *x*-axes show time.

The synthetic failure trace parameters used in Figure 5.8 are configured to be similar to the PlanetLab trace. In particular, the average server lifetime and the median downtime are the same. The result is an approximation (for example, PlanetLab grew during the trace) but the observed performance is similar. Some of the observed differences are due to batching (used by algorithm

described in Section 5.2 and Total Recall [BTC+04]) and timeouts (used by all systems); the impact of these are discussed in more detail in Sections 5.3.4 and 5.1.2.

Since the oracle system responds only to disk failures, it uses the lowest amount of bandwidth. The line labeled Cates shows a system that keeps track of exactly r_L replicas per object; this system approximates the behavior of DHTs like DHash, PAST and OpenDHT. Each failure causes the number of replicas to drop below r_L and causes this system to create a new copy of an object, even if the failure was transient. If the replica comes back online, it is discarded. This behavior results in the highest traffic rate shown. The difference in performance between the PlanetLab and Poisson trace is due to differences in the distribution of downtimes: Poisson is not a particularly good fit for the PlanetLab downtime distribution.

Total Recall [BTC+04] tracks up to a fixed number of replicas, controlled by a parameter r_H; we show $r_H = 5$ which is optimal for these traces, and $r_H = 9$. As can be seen at the right of the graphs, this tracking of additional replicas allows Total Recall to create fewer replicas than the Cates system. When more than r_L replicas are available, a transient failure will not cause Total Recall to make a new copy. However, Total Recall's performance is very sensitive to r_H. If r_H is set too low, a series of transient failures will cause the replication level to drop below r_L and force it to create an unnecessary copy. This will cause Total Recall to approach Cates (when $r_H = r_L$). Worse, when the system creates new copies it forgets about any copies that are currently on failed servers and cannot benefit from the return of those copies. Without a sufficiently long memory, Total Recall must make additional replicas. Setting r_H too high imposes a very high insertion cost and results in work that may not be needed for a long time.

Carbonite reintegrates all returning replicas into the replica sets and therefore creates fewer copies than Total Recall (and algorithm presented in Section 5.2). However, Carbonite's inability to distinguish between transient and disk failures means that it produces and maintains more copies than the oracle based algorithm. This is mainly visible in the first weeks of the trace as Carbonite builds up a buffer of extra copies. By the end of the simulations, the rate at which Carbonite produces new replicas approaches that of the oracle system.

5.3.3 How many replicas?

To formalize our intuition about the effect of extra replicas on maintenance cost and to understand how many extra replicas are necessary to avoid triggering repair following a transient failure, consider a simple Bernoulli process measuring R, the number of replicas available at a given

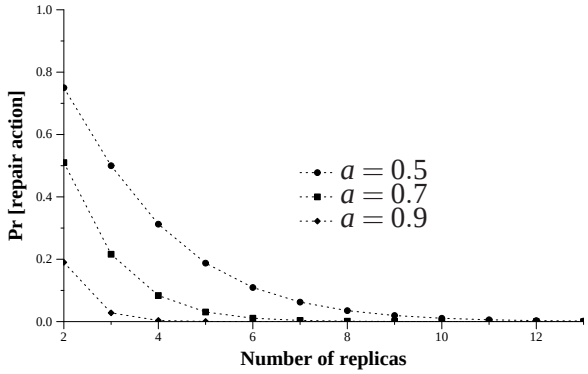


Figure 5.9: Additional redundancy must be created when the amount of live redundancy drops below the desired amount (3 replicas in this example). The probability of this happening depends solely on the average server availability a and the amount of durable redundancy. This graph shows the probability of a repair action as a function of the amount of durable redundancy, with $a = 0.5$, $a = 0.7$ and $a = 0.9$ for a replication system.

moment, when there are $r > r_L$ total replicas. The availability of each server is a. Since repair is triggered when the number of available replicas is less than r_L, the probability that a new replica needs to be created is the probability that less than r_L replicas are available:

$$\Pr[R < r_L \,|\, r \text{ extant copies}] = \sum_{i=0}^{r_L-1} \binom{r}{i} a^i (1-a)^{r-i}.$$

This probability falls rapidly as r increases but it will never reach zero; there is always a chance that a replica must be created due to a large number of concurrent failures, regardless of how many replicas exist already. However, when a large number of replicas exists, it is extremely unlikely that enough replicas fail such that fewer than r_L are available.

By computing the Chernoff bound, it is possible to show that after the system has created $2r_L/a$ replicas, the probability of a new object creation is exponentially small. $2r_L/a$ is a rough (and somewhat arbitrary) estimate of when the probability of a new object creation is small enough to ignore. Figure 5.9 shows (on the y-axis) the probability that a new object must be created when an increasing number of replicas already exist. As r increases, the probability that a new replica needs to be created falls, and the algorithm creates replicas less frequently. As r approaches $2r_L/a$, the algorithm essentially stops creating replicas, despite not knowing the value of a.

This benefit is obtained only if returning replicas are reintegrated into the appropriate replica set, allowing more than r_L to be available with high probability. As a result, the cost of responding to transient failures will be nearly zero. Still, this system is more expensive than an ora-

cle system that can distinguish between disk and transient failures. While the latter could maintain exactly r_L replicas, the former has to maintain approximately $2r_L/a$. The factor of $2/a$ difference in the cost is the penalty for not distinguishing disk and transient failures.

5.3.4 Create replicas as needed

Given that the system tends towards creating $2r_L/a$ replicas in order to keep r_L of them available, it is tempting to create the entire set—not just r_L of them—when the object is first inserted into the system (Total Recall [BTC+04] and algorithm presented in Section 5.2.1 use a similar technique). However, this approach requires an accurate estimate for a to deliver good performance. If a is overestimated, the system quickly finds itself with less than r_L replicas after a string of transient failures and is forced to create additional copies. If a is underestimated, the system creates unneeded copies and wastes valuable resources. Carbonite is simplified by the fact that it does not need to measure or estimate a to create the "correct" number of replicas.

Another idea is to create not only enough copies to bring the number of available replicas back up to r_L, but also e additional copies beyond r_L (this is similar to the algorithm described in Section 5.2 and Total Recall's lazy repair technique). Creating a batch of copies makes repair actions less frequent, but at the same time, causes more maintenance traffic than Carbonite. The work required to create additional replicas will be wasted if those replicas are lost due to disk failures before they are actually required. Carbonite, on the other hand, only creates replicas that are necessary to keep r_L replicas available. In other words, either Carbonite would eventually create the same number of replicas as a scheme that creates replicas in batches, or some replicas created in the batch were unnecessary: batch schemes do, at best, the same amount of work as Carbonite.

Figure 5.10 shows the bytes sent in a simulation experiment using a five-year synthetic trace with $a = 0.88$, $r_L = 3$, and an average server lifetime of one year. The graph shows results for different values of e (in Total Recall, $e = r_H - r_L$) and for two different scenarios. In the scenario with reintegration, the system reintegrates all replicas as they return from transient failures. This scenario represents the behavior of Carbonite when $e = 0$ and causes the least traffic.

In the scenario without reintegration, replicas that are unavailable when repair is triggered are not reintegrated into the replica set even if they do return. Total Recall behaves this way. Extra replicas give the system a short-term memory. Additional replicas increase the time until repair must be made (at which time failed replicas will be forgotten); during this time failed replicas can be reintegrated. Larger values of e give the system a longer memory but also put more data at risk of

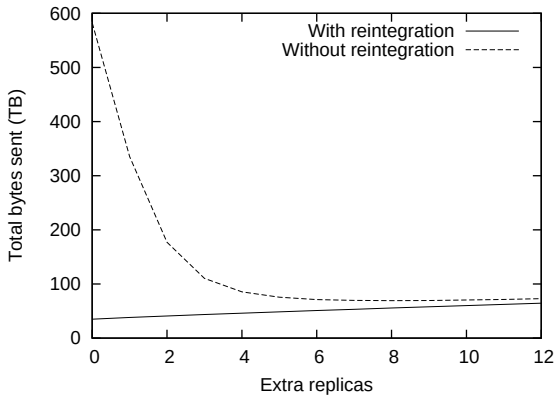


Figure 5.10: Total repair cost with extra replicas, and with and without reintegration after repair. Without reintegration, extra replicas reduce the rate at which repair is triggered and thus reduce maintenance cost; there is an optimal setting (here $e = 8$). With reintegration, the cost is lowest if no extra replicas are used.

failure: for this synthetic trace, a value of $e = 8$ is optimal. Taking advantage of returning replicas is simpler and more efficient than creating additional replicas: a system that reintegrates returning replicas will always make fewer copies than a system that does not and must replace forgotten replicas.

More formally, consider a replication based system with a perfect failure detector that makes E additional replicas when the number of replicas falls below a threshold r_L. let $r(t)$ be the number of replicas over time. We assume that $\frac{d}{dt}r = -\lambda r$ for some decay rate λ, i.e. the more data the system stores, the more data loss is expected per unit time. Thus, if the system creates $E \geq 0$ extra replicas when repair is triggered,

$$r(t) = (r_L + E) \cdot e^{-\lambda t},$$

assuming that the steady state or initial conditions has produced r_L copies. From this, we can derive the inter-repair time $T(E)$ by solving $r(T) = r_L - 1$ for T. We get

$$\begin{aligned} (r_L + E) \cdot e^{-\lambda T} &= r_L - 1 \\ T(E) &= \frac{1}{\lambda} \cdot \ln \frac{r_L + E}{r_L - 1} \end{aligned}$$

Thus, as E increases, $T(E)$ only increases logarithmically.

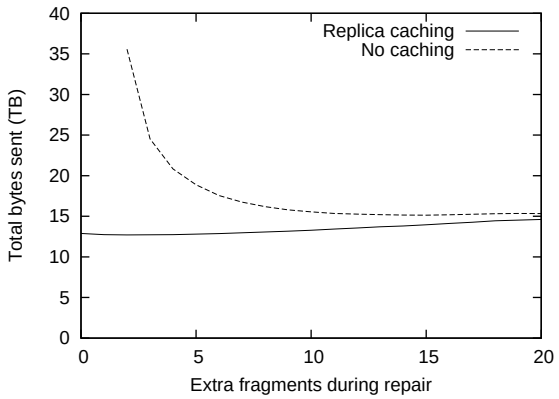


Figure 5.11: Total repair cost with a rate $r = \frac{m}{n} = \frac{7}{14}$ erasure-coding scheme, reintegration, extra fragments, and with and without replica caching after reconstruction and repair. Without caching, extra fragments reduce the rate at which repair is triggered and thus reduce maintenance cost; there is an optimal setting (here $e = 12$). With caching, the cost is lowest if few extra fragments are used ($e = 0$ to 2).

This affects the amount of work done per unit time, which we can view as the average number of replicas created per unit time. During each repair cycle, we create $1+E$ replicas, so the average number of replicas created per unit time is

$$\frac{1+E}{T(E)} = \frac{\lambda \cdot (1+E)}{\ln \frac{r_L+E}{r_L-1}}$$

which is minimal if $E = 0$ since the numerator increases faster than the denominator.

When the failure distribution is not exponential, on PlanetLab for example, creating one replica at a time is optimum assuming that all replicas created and available are tracked and reintegrated into the replica set (Figure 5.10).

5.3.5 Reintegration and Erasure-coding

For systems that use erasure codes, there is an additional read cost since a complete copy of the object is needed in order to generate a new fragment [RL05, WK02]. The cost of reading a sufficient number of fragments prior to recreating a lost fragment can overwhelm the savings that erasure codes provide. A common approach is to amortize this cost by batching fragment creation but simply caching the object at the server responsible for repair is much more effective. Figure 5.11 shows a simulation contrasting both caching and batching (but both with reintegration). Results in Figure 5.11 are similar to Figure 5.10: reintegrating and no batching is optimum. Caching the object

with a $r = \frac{m}{n} = \frac{7}{14}$ erasure code uses 85% of the bandwidth that the optimal batching strategy would use.

5.4 Summary

Repair, the second key to ensuring durability, must create replicas in response to failures. The problem of triggering repair is complicated by transient failure, where data is intact on a server, but not immediately available. In this chapter we showed that monitoring techniques cannot distinguish between permanent and transient failure and that costs due to transient failures were dominant in maintaining durability. We demonstrated three techniques to reduce costs due to transient failures.

First, we showed that timeouts reduce the number of transient failures by delaying classifying a server as failed. The effectiveness of timeouts in reducing false-positives, misclassifying servers as permanently failed that have actually only transiently failed, is dependent on the downtime distribution. Thus, if the timeout is set to mask most of the downtime distribution, a transiently failed server may return before a timeout expires and prevent resources from being wasted creating replicas unnecessarily. However, setting longer timeouts decreased durability because the time to recognize permanently failed servers increased, thus increasing the "window of vulnerability".

Second, we showed that extra replicas, beyond what was required to maintain a target durability, decreased costs due to transient failures. We demonstrated that the number of extra replicas required to minimize costs could be estimated. The advantage of extra replication is that it exponentially reduced the cost due to transient failures with a linear increase in replicas. Furthermore, there was no reduction in durability by adding extra replicas. As a result, extra replicas perform better than long timeouts.

Finally, we showed that the optimum number of extra replicas did not need to be estimated. Costs were minimized by simply responding to all failures, transient or permanent, creating replicas until r_L were available, tracking all replicas created, and reintegrating replicas after transient failure. We showed that this solution, called the Carbonite algorithm [CDH+06], created the minimum number of replicas, $2r_L/a$ (without an estimate for server availability a). The factor increase of $2/a$ was the cost for not being able to distinguish between permanent and transient failures. The result of the algorithm was that the system performs some extra work for each object early in its life, but over the long term creates new copies of the object only as fast as it suffers permanent failures.

Part III

Exploiting a Secure Log for Wide-Area Distributed Storage

Chapter 6

Secure Log Overview

In previous chapters, we discussed how to durably maintain data. In this part of the thesis, we design a distributed wide-area on-line archival storage system that employs durability techniques presented earlier. Additionally, the system solves the integrity problem. It ensures that the state of data stored in the system always reflects changes made by the owner.

We assume that an on-line archival storage system is an essential layer for a variety of applications and proceed to address two questions: First, what is an appropriate *interface* between applications and an archival infrastructure? Second, how can an archival infrastructure be constructed to provide integrity and efficiency in addition to durability?

Our basic premise is that a secure log provides an ideal primitive for implementing an archival storage infrastructure. A log's structure is simple and its security properties can be verified [LKMS04, MRC+97, Mer88, MMGC02, SK98]. Only a single interface, `append()`, is provided to modify the log, and all mutations occur at a single point—the log head. A system can secure the log head by requiring that all `append()` operations be signed by the private key of the log owner. If each log element is named individually, random access `get()` provides quick data retrieval. Because of the simplicity of its interface, a secure log is easier to implement in a secure way than other structures. In particular, only a narrow interface that modifies data needs to be secured. Additionally, most of a secure log is resistant to corruption or attack since it is immutable (read-only and cannot change). Furthermore, a secure logs interface is sufficient to implement a variety of interesting applications. For instance, we demonstrate that a secure log interface can be used to implement a secure file system application.

In the following chapters, we show how to construct an efficient Byzantine fault-tolerant wide-area archival system with a secure log interface. Such an archival system is intended to be a

component of a larger application. While a secure log is conceptually simple, replicating the log in a distributed storage system has proved challenging [MMGC02, REG+03]. Such systems offer improved data durability, but must overcome several disadvantages arising from the distributed environment. We address the challenges of consistency, durability, and efficiency.

When a system replicates data, it must ensure that replicas are kept consistent and queries are answered in a manner that reflects the true state of the data. Maintaining consistency of a log is simpler than other data structures because modifications affect only the log head. In particular, most of the secure log is immutable where consistency is immediate. Still, a system must maintain the consistency of the log head as servers hosting the replicas fail or endure attacks. To ensure progress, the system must manage the replica set, replacing faulty servers with new ones.

Replication alone does not ensure durability. A system must also respond intelligently to changes such as the scheduled retirement of old servers and unexpected transient or permanent failures. Furthermore, the system must tolerate and recover from a variety of faults and attacks. For instance, data may be corrupted on disk or during network transmission and malicious agents may attempt to subvert the system. The system should persist despite server and network failure.

Finally, a wide-area storage system must utilize system resources carefully. Protocols should limit the number of cryptographic operations and the amount of communication needed across the wide-area. This requirement leads to aggregation—combining small, application-sized blocks into larger chunks for validation, storage, and repair. Aggregation is notably lacking from recent DHT-based storage systems [BTC+04, DKK+01, MGM04, MMGC02] that divide large objects into small (e.g. 8K) blocks which are spread widely.

This part of the thesis describes the design of a secure, distributed, log-structured storage system. To meet the challenges described above, the design is based on a log interface. The system combines this interface with three technologies: quorums, quorum repair, and aggregation. We utilize dynamic Byzantine fault-tolerant quorums to ensure consistency of the log head. Data integrity is assured at both the block and container granularity. We provide data durability with an algorithm that repairs quorums when replicas fail. Finally, aggregation reduces communication costs while maintaining fine-granularity access for clients.

The contributions of this part of the thesis are as follows:

- An implementation of a secure log interface.
- A consistency protocol for a replicated secure log based on dynamic Byzantine fault-tolerant quorums that works well in the wide-area.

- A mechanism for dynamically repairing Byzantine fault-tolerant quorums that maintains consistency and durability in the face of recurring replica failure.
- In Part IV, we describe an operational prototype that combines these features and is currently running in the wide-area.

In the rest of this section, we give an overview of goals, models, and assumptions of a system based on a secure log interface.

6.1 Overview

A secure log is a generic low-level data structure and interface used by distributed wide-area storage systems to provide secure, durable storage. It is designed to serve as the storage layer for a variety of applications such as file systems [DKK+01, MMGC02], back-up [QD02, REG+03], and databases[1]. It provides to applications a limited interface by which they can create new logs, append data to the head of an existing log, and read data at any position in the log. It can be used to guarantee fault-tolerance through replication, consistency via dynamic Byzantine fault-tolerant quorum protocols, and efficiency by aggregation.

6.1.1 Storage System Goals

The design of a storage system based on a secure log was guided by the following goals.

- Integrity: Only the owner can modify the log. Any unauthorized modifications to the log, as in substitution attacks, should be detected.
- Incremental Secure Write and Random Read Access: A client can add data to a log securely as it is created, without local buffering. Further, the client can read arbitrary blocks without scanning the entire log.
- Durability and Consistency: The log should remain accessible despite temporary and permanent server failure. The system should ensure that logs are updated in a consistent manner.

[1] A secure log inherently supports transactional databases as an underlying storage layer since it stores data using ACID (Atomic, Consistent, Isolation, and Durable) semantics: all writes are atomically applied to the log and stored with a total order within the log structure.

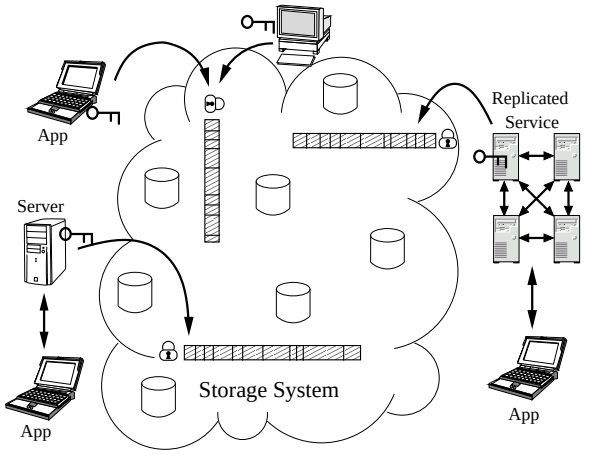


Figure 6.1: A log-structured storage infrastructure can provide storage for end-user clients, client-server systems, or replicated services. Each log is identified by a key pair.

- Efficiency/Low overhead: Protocols should limit the number of cryptographic operations and the amount of communication needed across the wide-area. The infrastructure should amortize the cost of maintaining data and verifying certificates when possible.

A system that provides these goals would be a useful substrate for applications. Integrity ensures that data stored in the system reflects changes made by the application. For usability, applications can write new data and read any block already written. Durability and consistency ensure that the log will exist and be writable even as servers fail. Finally, applications can efficiently use the storage system reducing the number of cryptographic and wide-area operations.

6.1.2 System Model

The storage system stores logs on behalf of clients. The types of clients storing data in the system can vary widely as shown in Figure 6.1. The client may be the end-user machine, the server in a client-server architecture, or a replicated service. In any case, the storage system identifies a client and its secure log by a cryptographic key pair; only principals that possess the private key can modify the log. Requests that modify the state of the log must include a certificate signed by the principal's private key. Although a log is non-repudiably bound to a single key pair, multiple instances of the principal may exist simultaneously. If multiple devices possess the same private key, then they can directly modify the same log.

Storage resources for maintaining the log are pre-allocated in chunks. When a new chunk, or *extent*, needs to be allocated, the system consults the *administrator*. The administrator authenti-

cates the client needing to extend its log and selects a set of storage servers to host the extent. The newly-allocated portion of the log is replicated on the set of selected storage servers. To access or modify the extent, clients interact directly with the storage servers.

Applications interact with the log through a client library that exports a thin interface—`create()`, `append()`, and `get()`. To create a new log, a client obtains a new key pair and invokes the `create()` operation. The administrator authenticates the request and selects a set of servers to host the log.

After a log has been created, a client uses the `append()` operation to add data to the head of the log. The client library communicates directly with the log's storage servers to append data. The interface ensures that data is added to the log sequentially by predicating each write on the previous state of the log. If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log[2].

Data written to a log cannot be explicitly deleted. Instead, implicit deletion based on an expiration time is supported. A client can extend the expiration time of an extent.

6.1.3 Assumptions

We assume that clients follow specified protocols, except for crashing and recovering. A malfeasant client, whether due to software fault or compromised key, can prevent the system from appending data to a log. It cannot, however, affect data already stored in its log or logs belonging to other principals. If a principal's private key should be compromised, an attacker could append data to the log, but it cannot destroy data previously stored in the log. A principle can retrieve data from a log until the log's expiration time.

We assume that the administrator, tasked to select sets of storage servers to host logs [MA04], is trusted and non-faulty. The design, however, includes several mechanisms to mitigate the cost and consequences of this assumption. While each log uses a single administrator, different logs can use different administrators. By allowing multiple instances, the role of the administrator scales well. Second, the administrator's state can be stored as a secure log in the system. Thus, the durability of the state can be assured like any other log. If the administrator were to fail, a new administrator could be created using the state stored in the log. Third, the state of the administrator can be cached to reduce the query load on an administrator. Finally, the administrator can be implemented as a replicated service to improve availability further.

[2] We assume a storage server atomically handles each request. That is, a server processes requests one at a time, even though multiple requests may have been received at the same time.

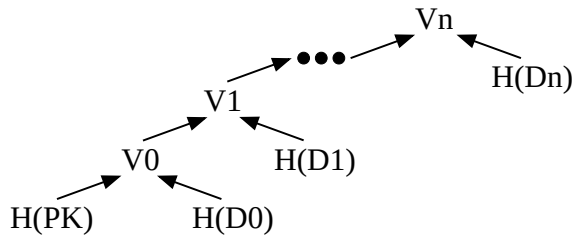


Figure 6.2: To compute the verifier for a log, the system uses the recurrence relation $V_i = H(V_{i-1} + H(D_i))$. $V_{-1} = H(PK)$ where PK is a public key.

Storage servers may exhibit Byzantine faults. We assume that, in the set of storage servers selected by the administrator to host a particular extent, a maximum threshold number of servers is faulty.

6.2 Secure Log Details

A secure log is a data structure with interesting properties and an interface that protects its properties. First, only a single operation, `append()`, can modify the log. Most of the log, except for the log head, is immutable – read-only and cannot change. Second, a single cryptographically secure hash, called the *verifier*, asserts both the data and append-order integrity of the entire log. The verifier is cheap to compute and maintain over time. Third, consistency of the log is assured by requiring the verifier of the previous state of the log as a predicate in a subsequent `append()` operation. Finally, since each log element is individually named, random access `get()` provides quick data retrieval. We discuss the verifier, `append()`, and `get()` in more detail below.

First, the data and append-order integrity of the entire log is assured via Merkle's hash chaining technique [Mer88]. This technique works by naming each element in the log with a cryptographically secure hash of the content of the element and embedding the secure name in the data structure. With Merkle chaining, a malicious or compromised server cannot deceive a client with corrupt data. Further, Merkle's technique makes the log self-verifying meaning the integrity of the entire log can be verified with a single hash called a *verifier*.

A verifier is computed as follows. Assume a log contains a sequence of variable-sized data blocks, D_i. Each data block is named with a secure, one-way hash function, $H(D_i)$. The verifier is computed using the recurrence relation $V_i = H(V_{i-1} + H(D_i))$, where $+$ is the concatenation operator. We bootstrap the process by defining V_{-1} to be a hash of the public key that identifies the log. See Figure 6.2. This convention ensures that logs owned by different principals always have

Certificate contents:	
verifier	token that verifies contents of log
seq_num	certificate sequence number
timestamp	creation time of certificate
ttl	time the certificate remains valid

Table 6.1: The certificate present with each operation and stored with each log. It includes fields to bind the log to its owner and other metadata fields.

different verifiers.

Creating a verifier in this manner has several advantages. When a block is appended to the log, the client can compute the verifier incrementally. This means it must hash only the new data, not all data in the log, to compute the running verifier. Additionally, only *one* particular sequence of appends result in a particular verifier. Thus, chaining creates a verifiable, time-ordered log recording data modifications. Furthermore, requiring the latest verifier as a predicate in subsequent `append()` operations assures servers maintain a consistent state of the log. A server atomically performs an `append()` against its locally stored state. If the predicate matches the currently stored verifier, then the server applies the `append()`; otherwise, the `append()` is rejected.

To `append()` data to the log, a client creates a request and submits it to the storage servers. A request has three arguments. (1) A predicate – verifier that securely summarizes the current state of the log. (2) New data to append to the log. And (3) a new certificate that includes a new verifier and new sequence number. The certificate verifier summarizes the next state of the log after appending data. The sequence number is a monotonically increasing number. Table 6.1 shows the contents of a certificate.

When a server receives an `append()` request, it determines if a request succeeds or not. It performs several checks using local knowledge. The certificate contained in the request must include a valid signature. Also, the predicate verifier contained in the request must match the current state of the log recorded by the storage server. Additionally, the verifier in the certificate must match the new verifier after appending new data to the log. Further, the sequence number in the certificate must be greater than the one currently stored. If these conditions are met, the server writes the new data to the log on its local store and returns success to the client. Otherwise, the request is rejected and failure is returned.

To `get()` data, the client library must first locate the server storing the requested block(s) and then retrieve the block(s) from that server. If the entire log was stored by one server, then the client could retrieve the requested block(s) from that one server. However, without a limit on the

Log Interface:		
status	=	*create*(H(PK), cert);
status	=	*append*(H(PK), cert, predicate, data[ ]);
data[]	=	*get_blocks*(extent_name, block_name[ ]);
mapping	=	*get_map*(extent_name);

Table 6.2: Operations to `create()`, `append()`, and retrieve data via `get_blocks()` from a secure log. A log is identified by the hash of a public key ($H(PK)$). The `create()` and `append()` operations include a certificate. Further, `append()` requires a verifier of the previous state of the log as a predicate. The `get_blocks()` operation requires two arguments because the system breaks logs into extents and requires both the extent_name and block_name. The `get_map()` retrieves the mappings of a previous extent_counter to previous extent_name.

number of `append()`'s that can be performed on a log, the size of the log can grow boundlessly large. As a result, the storage system stores a log not as a sequence of log elements, but rather a sequence of container objects called *extents*. Extents are the units of storage and are independently maintained by the storage system. Each extent stores a variable number of arbitrarily-sized application-level log elements (blocks). Additionally, similar to the log itself, extents are self-verifying and use a verifier to guard data and order-integrity. The use of extents to aggregate log elements into larger containers was first proposed by Eaton et al. [EWK05]. We describe an implementation of extents as part of a secure log in Chapter 7.

Extents introduce added complexity in computing the address of a block of data. Each extent is assigned an integer corresponding to its position in the chain. Also, each extent records the mapping between counter and extent name for the previous extent.

To read data embedded in extents, the client must know the extent name, location of server storing extent, and block name. However, the application only records the extent counter and block name; it does not know the extent name at the time of `append()`. As a result, the client library must first resolve the extent counter to extent name. In particular, the client library first accesses the mappings stored in the log head via `get_map()` to determine the previous extent name. We assume that a mechanism exists to locate the log head or any extent given the extent name (Chapter 9). Next, the client continues retrieving and resolving extent counter to extent name mappings until it locates the mapping that includes the desired extent holding the data. It then uses the `get_blocks()` operation to retrieve the requested blocks from that extent. To accelerate the translation between counter and extent name, the client library caches the mappings. Also, in implementation, each extent contains not just the mapping for the previous extent, but a set of mappings that allow resolution in a logarithmic number of lookups.

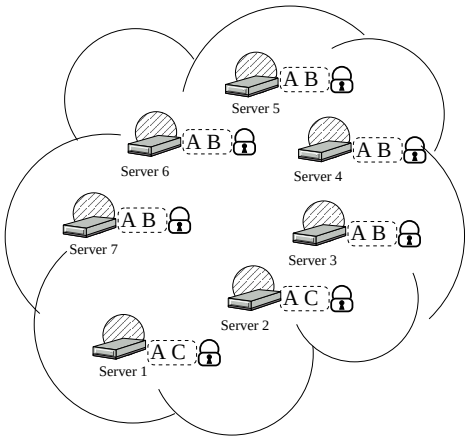
Table 6.2 summarizes the log interface: `create()`, `append()`, and `get()` (`get_map()`and `get_blocks()`).

6.3 Semantics of a Distributed Secure Log

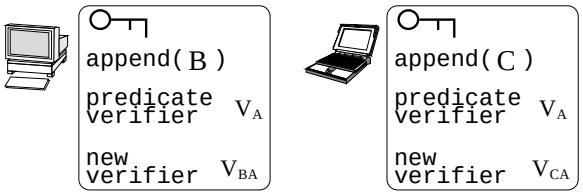
The archival system replicates the log on multiple servers to provide durability. Durability means that the log persists over time. The difficulty is maintaining consistency across the log replicas so that new data can be added. The storage system should be capable of maintaining data and append-order integrity of the log across the replicas despite arbitrary failure such as network error, server failure, or simultaneously submitted and conflicting requests. As a result, consistency across the log replicas must be maintained to ensure progress – ability to add new data to the log. We discuss a consistency protocol in Chapter 8; in this section, however, we describe the client's view considering that the client interacts with multiple servers to complete a single operation.

An operation that modifies the log results in one of three states: sound, unsound, or undefined. The result of an operation is *sound* if the client receives a positive acknowledgment from a threshold of servers. A request succeeds and is "durable" if the result is sound. Durable means that data exists over time in the storage system even as servers fail. On the other hand, the result of an operation is *unsound* if the client receives a negative acknowledgment from enough servers such that positive acknowledgment from a threshold is no longer possible (e.g. sizeof(negative acks) $\geq$ sizeof(server set) $-$ threshold$+1$). A request fails if the result is unsound. The storage system does not maintain unsound results, thus unsound writes are not durable. Finally, the result is *undefined* if it is neither sound nor unsound. An undefined result means the client did not receive sufficient acknowledgment from servers perhaps due to network or server failure. In this case of an undefined result, a timeout occurs and the client does not know whether the request is sound or unsound. After a timeout, the client performs a `get_cert()` on all the servers and waits to receive acknowledgment from a threshold. If the state stored in the system has changed (another client updated the log), then the request is unsound. If the `get_cert()` fails to receive acknowledgment from a threshold of servers, then the client may trigger a repair audit that will determine the latest consistent state of the log (repair audits are discussed in Chapter 8). The client continually sends the request, reads the state of the system, and then triggers a repair audit until the request is either sound or unsound.

To illustrate the notions of sound, unsound, and undefined writes, assume a log is replicated on seven servers. A threshold required for consistency and a sound response is five positive acknowledgments. The number required for an unsound response is three negative acknowledgments



(a) Storage System



(b) Request Succeeds (c) Request Fails

Figure 6.3: Semantics of a Distributed Secure Log. (a) A secure log with the value *A* is initially replicated onto seven servers. In (b), a workstation attempts to `append()` the value *B*, predicated on *A* already being stored. The result of the request is sound since it reaches a threshold of servers (servers 3-7). In (c), a laptop, which possesses the same private key as the workstation, simultaneously attempts to `append()` value *C*, predicated on *A* already being stored. The result of the request is unsound since the predicate fails on a threshold of servers. Note that the two servers (server 1-2) apply *C* since the predicate matches local state. However, the system should return value *B* in any subsequent reads.

(total minus a threshold plus one, $7-5+1=3$). The initial value stored on all the log replicas is *A*. Further, assume two clients, a workstation and laptop, simultaneously submit conflicting operations. The workstation attempts to append the value *B* and receives five positive acknowledgments and two negative, thus the response is sound since a threshold acknowledged positively. The laptop, on the other hand, attempts to append the value *C* and receives five negative acknowledgments and two positive, thus the response is unsound. With this scenario, the storage system should maintain the workstation's appended value *B* over time despite arbitrary server failure. Figure 6.3 shows this example. Furthermore, in the above example, if the workstation receives one less positive acknowledgment (four instead of five), possibly due to network transmission error, then the result would be undefined and timeout. The workstation could read the latest replicated state of the secure log,

trigger a repair audit that will repair the distributed secure log if necessary, and resubmit the request until it receives sufficient server acknowledgment.

Alternatively, if both the workstation and laptop requests received unsound responses (e.g. both received three negative acknowledgments), then the log replicas would be in an inconsistent state since a threshold of the log replicas state do not agree. When the log replicas are in an inconsistent state no progress can be made, new data cannot be added to a threshold of the log replicas, and the log replicas need to be *repaired* to a consistent state. Repair restores the log replicas to a consistent state such that the latest sound write is the last write stored by a threshold of log replicas. A quorum repair protocol that ensures consistency and durability amongst log replicas is discussed in Chapter 8.

6.4 Example uses of a Secure Log

To understand how a client can use a log, consider the examples of a tamper-resistant syslog, secure file system, and a database log.

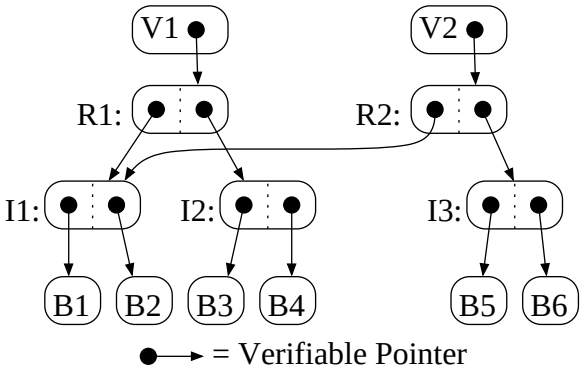
6.4.1 File System Interface

This example shows how a versioning file system stores data in a log shown in Figure 6.4.

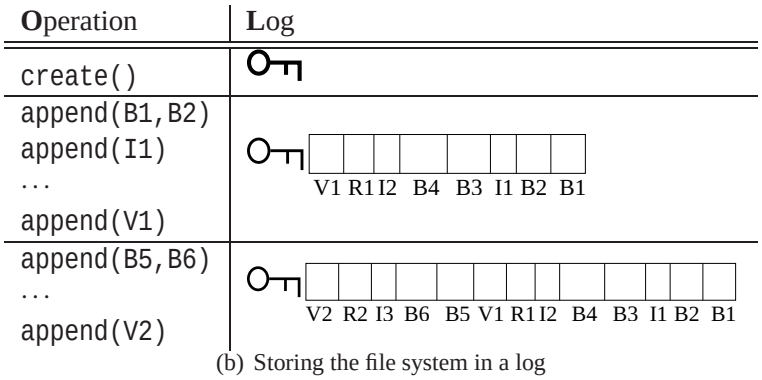
Figure 6.4(a) shows an abstract representation of two versions of a versioning file system. The second version is composed of some newly-written data and some data from the previous version. The application first translates the file system into a Merkle tree [Mer88]. It divides the files and directories into small blocks, typically 4–8 KB or less. Each block is named by a secure hash of its contents. Applications can embed the secure pointers in application-level data to create complex data structures [DKK+01, REG+03, QD02].

To store the file system into the log, the application uses the `create()` interface to initialize the log. It then traverses the Merkle tree in a depth-first manner, using the `append()` operation to write data to the log. Figure 6.4(b) shows the operations and the contents of the log.

To read data, the application invokes a `get_blocks()` operation with the name of the desired block. By naming blocks individually, the interface supports random reads. The secure pointers that name blocks also allow the application to verify data integrity. The application simply compares the hash of the data retrieved from the log against the name by which it was retrieved. The application verifies all data read by following chains of secure pointers.



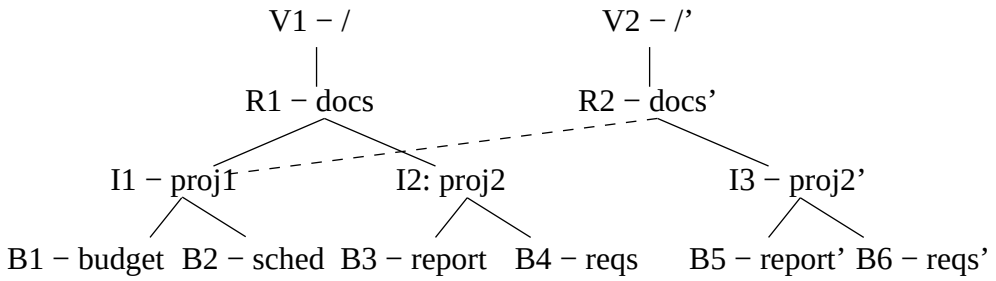
(a) A versioning file system



(b) Storing the file system in a log

Figure 6.4: (a) An abstract representation of a versioning file system with two versions. A version can reference newly created data and data in previous versions. (*V* = version, *R* = root directory, *I* = file inode, *B* = data block) (b) An application can write the file system to a log by traversing the tree in a depth-first manner.

To make this file system example concrete, we map the block symbols to files and directories. We assume that a file or directory is stored as a single block and ignore inodes in this example. In Figure 6.5, V*x*, R*x*, I*x*, and B*x* represent the root "/" directory, "docs" directory, "proj" directories, and files, respectively. A single quote (') represents new versions of a directory or a file. To write new data, the file system appends the changed files and directories to the head of the log in a depth first manner. For example, when the file system application wrote new versions of the report and reqs documents (B5–report' and B6–reqs'), it appended the new file versions to the head of the log by calling `append(B5-report')` and `append(B6-reqs')` Additionally, the new directory versions that point to the new files are appended to the log (`append(I3-proj2')`, `append(R2-docs')`, and `append(V2-/')`). To read a particular file, the file system application reads the root of the file system stored at the head of the log and follows the pointers to the desired file. For example, assume the client wants to read the "sched" file. The client first reads the root of



(a) First version of file system (b) Second version of file system

Figure 6.5: A simple file system used as a running example. Map symbols to concrete file system.

the file system which is the first entry stored at the head of the log, `get_head()`. The root of the file system is a directory and directories store pointers to children directory and files. Pointers are an extent counter and secure hash. The file system application uses the `get_map()`routine to map the extent counter to extent name and caches the mapping. After resolving the mapping, the file system reads the next directory calling `get_blocks(extent_name, H(R2-docs’))`. Similarly, the file system does the same for “proj1” and “sched”. It extracts the pointer from the parent directory, maps the extent counter to extent name, then calls `get_blocks(extent_name, H(I1-proj1’))` and `get_blocks(extent_name, H(B2-sched’))`.

6.4.2 Database Example

Similar to the file system example, a database can store data in a secure log. For example, Figure 6.4 could also represent a database where data is stored as a B-tree. `append()` is used to add data to the log after modifying or adding new entries. Similar to the file system example, the pointers to database nodes and entries are extent counter and hash of the entry. Finally, `get_head()`, `get_map()`, and `get_blocks()` can be used to retrieve any block.

6.4.3 Tamper-resistant syslog

As a last example, a tamper-resistant syslog can be used by an operating system to store all access entries, which can be audited at a later date. The secure log ensures that entries have not been altered or deleted.

Chapter 7

The Secure Log Interface

The `create()`/`append()` interface should assure the integrity of data stored in a log. The interface is sufficient to assure integrity when an entire secure log is stored unreplicated on a single server. It ensures data is appended to the log in a sequential fashion. It ensures the append-order and data are verifiable. It ensures only entities that possess the private key can `append()` data. It, however, is not sufficient to ensure the durability of the log (the single server could permanently fail).

For durability, the secure log is replicated with replicas distributed throughout the wide-area. Furthermore, the entire secure log is not stored together, but rather broken into a sequence of containers called *extents*. Extents are the units of replication and storage. Even though a secure log is replicated and broken into extents, its interface should still ensure the integrity of the log as if it were stored whole, unreplicated, and on a single server (i.e. data is appended to the log in a sequential fashion, the `append()` order is verifiable, and only entities that possess the private key can `append()` data).

Given that a secure log is replicated and broken into extents, the challenge is implementing a secure log interface that can be used to ensure both the durability and integrity of a distributed secure log. We define such an interface in this chapter and show how a client library uses it to interact with the storage system.

The rest of this chapter describes in more detail extents, the secure log interface, and distributed secure logs. In Section 7.1 we discuss background and prior work. Next, in Section 7.2, we describe how to use an extent aggregation interface to construct a secure log. We describe the complete distributed secure log interface in Section 7.3. Finally, in Section 7.4, we discuss why this interface is easier to implement in a secure way than others.

7.1 Background and Prior Work

In this section, we begin by reviewing the concepts behind self-verifying data. We then discuss the designs of popular, first-generation distributed hash table (DHT) storage systems, focusing on their similarities and the consequences of those decisions.

7.1.1 Self-verifying Data

Data is said to be self-verifying if it is named in a way that allows any client to validate the integrity of data against the name by which it was retrieved. Therefore, names of self-verifying data, serve ideally as identifiers in a distributed wide-area storage system. The self-verifying property enables clients to request data from any machine in the network without concern of data corruption or substitution attack. A malicious or compromised machine cannot deceive a client with corrupt data—its attack is limited to denying a block's existence.

Traditionally, data is made self-verifying via one of two techniques: hashing and embedded signatures. These techniques were made popular by the Self-certifying Read-only File System [FKM00]. *Hash-verified data* is named by a secure hash of its content. A client can verify hash-verified data by computing the hash of the returned data and comparing it to the name used to fetch the data. Hash-verified data is immutable—if the data changes, the hash-verified name of the data changes too.

Additionally, Weatherspoon et al. extended the hash-based approach to name erasure code fragments in a self-verifying manner [WK02] as discussed in Section 4.1.2. Clients can verify either individual erasure code fragments or the full block of data by the same name. Distillation codes [KSL+04] can be considered a generalization of this scheme.

Key-verified data is verified via a certificate that is signed by a user's public key. The certificate contains some token, such as a secure hash of the content, that securely describes the data. To verify key-verified data, a client checks the signature on the certificate and compares the data against the verifier in the certificate. Commonly, key-verified data is named by a hash of the public key that signs the data's certificate. With this approach, each key pair can be associated with only a single object. To allow the system to associate multiple objects with a single key pair, other schemes hash a combination of data, such as the public key and a human-readable name, to create the name for the data. Key-verified data can be mutable—a client can associate new data with a key by creating a new certificate.

Many systems employ Merkle's chaining technique [Mer88] with hash-verified data to

Traditional interface:		
		put_hash(H(data), data);
		put_key(H(PK), data);
data	=	*get*(h);

Table 7.1: First-generation distributed hash table (DHT) storage systems use a simple `put()/get()` interface. The `put_hash()` and `put_key()` functions are often combined into a single `put()` function. $H()$ is a secure, one-way hash function; h is a secure hash, as output from $H()$.

combine blocks into larger, self-verifying data structures. Such systems embed self-verifying names into other data blocks as secure, unforgeable pointers. To bootstrap the process, systems often store the name of the root of the data structure in a key-verified block, providing an immutable name for mutable data. To update data, a client replaces the key-verified block. See, for example, CFS [DKK+01], Ivy [MMGC02], and Venti [QD02].

7.1.2 Distributed hash table (DHT) storage systems

Recently, researchers have used self-verifying data and the distributed hash table (DHT) technology as a foundation for building distributed wide-area storage systems. Despite their independent development, many systems share important design features. In identifying common design features, we have considered a number of popular, first-generation DHT storage systems in the research literature including CFS [DKK+01], Ivy [MMGC02], OceanStore [REG+03], Total Recall [BTC+04], and Venti [QD02].

First-generation DHT storage systems provide a simple interface for clients to interact with the storage system. The interface, shown in Table 7.1, is often called a `put()/get()` interface due to its similarity to the interface of a hashtable. Note, while we have shown `put_hash()` and `put_key()` as distinct members of the interface, they are often implemented as a single `put()` function.

Systems tend to use self-verifying data and the `put()/get()` interface in a common manner, illustrated in Figure 7.1. A client divides data into small blocks, typically 4–8 KB or less. It computes the hash-verifiable name of each block and links the blocks together, using the names as unforgeable references, to create a Merkle tree. Finally, the client stores all blocks of the tree in the DHT system using the `put_hash()` interface. If the system supports mutable data, the client will typically use the `put_key()` function to store a key-verified block that points to the root of the Merkle tree, providing an immutable name to the mutable data.

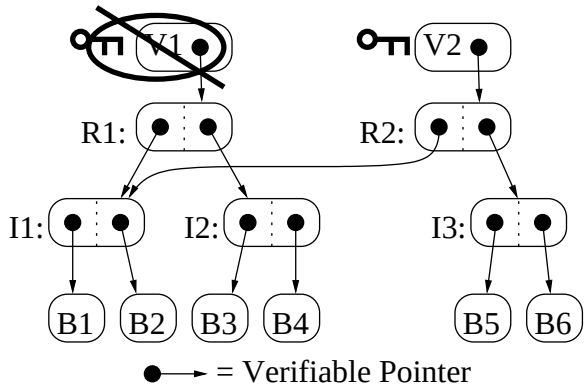


Figure 7.1: Clients divide data into small blocks that are combined into Merkle trees. A key-verified block points to the root of the structure. To update an object, a client overwrites the key-verified block to point to the new root. (*V* = version, *R* = version root, *I* = indirect node, *B* = data block)

To read data, a client first retrieves and validates the key-verified root block of the data structure using the `get()` function. It can then iteratively fetch and verify the other hash-verified blocks by following the chain of hash-verified names embedded in the tree.

Because each new hash-verified block of data has a unique name, DHT storage systems naturally provide versioning capabilities. Some systems expose the versioning feature to the end user [REG+03] while others do not. Using copy-on-write to provide efficient versioning has also been implemented in other systems predating DHT storage systems [MT85].

One notable counterexample to these design patterns is the PAST [DR01] system. PAST uses the `put_hash()` call to store whole objects as hash-verified blocks. As a result, PAST cannot incrementally update objects; instead, it stores new versions of an object as a single block using the `put_hash()` interface.

The design features shared among these implementations have a significant impact on the behavior of the resulting systems. For example, the `put()/get()` interface forces the storage infrastructure to manage data at the same granularity as the client. While some applications, like off-line data processing, handle data in large chunks, many interactive and user-oriented applications tend to create and access relatively small blocks of data. By supporting fine granularity access, these systems allow applications to fetch data without wasting scarce bandwidth at the edges of the network retrieving data that is not needed or already cached. It allows applications to push data to the infrastructure as soon as it is created, improving durability.

Coupling the infrastructure's unit of management with the client's unit of access, however, has several disadvantages. Most relevant to our work, because each block is managed independently

in the infrastructure, to provide non-repudiable binding of owner to data, a client must create a certificate for each block. To illustrate this problem, assume an application running on a 3 GHz processor wishes to store 1 TB of data. If the data is divided into 8 KB blocks and the certificates are created using 1024-bit RSA cryptography, it would take more than *six days* to create certificates for the data [1].

Other consequences, though secondary to our work, also impact the efficiency of the system. For example, some of the indexing, management, and maintenance costs in the infrastructure are independent of the block size. Thus, managing the small blocks created by the client increases the load on the infrastructure. Also, because each application-level block is indexed independently, clients must issue separate requests for each block they wish to read. Reading an object of even moderate size can flood the storage system with requests.

7.1.3 Prior Aggregation Systems

The classical file system literature demonstrates repeatedly how aggregation can improve efficiency of storage systems. For example, the Fast File System (FFS) [MJLF84] increases system performance, in part, by aggregating disk sectors into larger blocks for more efficient transfer to and from disk. XFS [ADN+95] further aggregates data into extents, or sequences of blocks, to reduce the size of the metadata and allow for fast sequential access to data. GoogleFS [GGL03] aggregates data from a single file further still into 64 MB chunks, improving performance and per-object maintenance costs for large files typical of their application domain.

More recently, the Glacier [HMD05] DHT storage system, has shown how aggregation can reduce the number of objects that the system must index and manage. Glacier [HMD05] relies on a proxy trusted by the user to aggregate application-level objects into larger collections. All collections in Glacier are immutable and thus cannot be modified after they are created.

7.2 How to use an Aggregation Interface to Construct a Secure Log

In this section, we summarize a method for aggregating variable-sized application-level blocks into ordered containers called extents. Furthermore, we show how to break a secure log into an ordered sequence of extents.

[1] A 3 GHz Pentium-class processor can create a signature in 4 ms, as measured with the command `openssl speed rsa1024`.

Breaking a secure log into extents serves two purposes. First, it allows a log to grow without requiring the entire log to be stored together. Extents are the units of storage and are independently maintained by the storage system. Second, extents aggregate log elements (blocks) together improving system efficiency. Each extent contains an ordered collection of variable-sized application-level blocks of data. Use of extents to aggregate blocks together into larger containers was first proposed by Eaton et al. [EWK05]; we summarize the aggregation interface and show how to construct a log from it.

A secure log broken into extents still maintains the integrity of the entire log. All data in an extent belongs to the same log, and thus, is owned by a single principal. The sequence of blocks within extents defines the append order of the extent. Moreover, the sequence of extents define the append order of the entire log. Blocks, extents, and the entire log are all self-verifying via secure hashes and Merkle chaining.

A log is composed of two types of extents. The log head is a mutable, key-verified extent; all other extents are immutable, hash-verified extents. The key-verified log head is named by a secure hash of the public key associated with the log. To verify the contents of the log head, a server compares the data to the verifier included in the certificate (after confirming the signature on the certificate). When the mutable extent at the log's head is full, the system converts the extent into an immutable hash-verified extent. A hash-verified extent is named by a function of the contents of the extent. Specifically, the extent is named by the verifier in the extent's most recent certificate. A server can verify the integrity of a hash-verified extent by comparing an extent's contents to its name (verifier).

7.2.1 Constructing a Secure Log

Table 7.2 shows the `create()`/`append()` secure log interface extended to include an extent interface. All mutating operations require a certificate signed by the client for authorization. The certificate includes the verifier of the new version of the extent. The interface ensures that updates are applied in a sequential manner by predicating each operation on the previous state of the extent. Upon completion of the operation, the certificate is stored with the extent. The `snapshot()` and `truncate()` operations help manage the chain of extents. The `put()` operation is an optimization that allows a data source to write data directly to a hash-verified extent. The `renew()` operation extends the expiration time of an extent.

Three of the operations enumerated in Table 7.2—`create()`, `snapshot()`, and `put()`—

Interface for Aggregation:		
status	=	*create*(H(PK), cert);
status	=	*append*(H(PK), cert, predicate, data[ ]);
status	=	*snapshot*(H(PK), cert, predicate);
status	=	*truncate*(H(PK), cert, predicate);
status	=	*put*(cert, data[ ]);
status	=	*renew*(extent_name, cert);
cert	=	*get_cert*(extent_name);
data[]	=	*get_blocks*(extent_name, block_name[ ]);
extent	=	*get_extent*(extent_name);
mapping	=	*get_map*(extent_name);

Table 7.2: To support aggregation of log data, we use an extended API. A log is identified by the hash of a public key ($H(PK)$). Each mutating operation must include a certificate. The `snapshot()` and `truncate()` operations manage the extent chain; the `renew()` operation extends an extent's expiration time. The `get_blocks()` operation requires two arguments because the system implements two-level naming. The `extent_name` is either H(PK) for the log head or verifier for hash-verified extents.

create new replicas. Each of these operations requires that the system contact the administrator for a configuration, set of servers, to host the new replicas. The most common operation, `append()`, does not require any interaction with the administrator.

While the application still relies on the simple `create()`/`append()` interface, the client library interacts with the storage system using the extended API. Figure 7.2 illustrates how the client library uses the extended API. In this example, an application is writing the first version of the file system shown in Figure 6.4 to the storage infrastructure. The client library passes most `append()` requests from the application to the storage system. Periodically, however, to prevent the extent at the log head from growing too large, the client library copies data to hash-verified extents using the `snapshot()` operation. After data has been copied to a hash-verified extent, the library uses `truncate()` to reset the log head. While `snapshot()` and `truncate()` are typically used together, we have elected to make them separate operations for ease of implementation. Individually, each operation is idempotent, allowing the library to retry the operation until successful execution is assured. The library continues to use the `append()`, `snapshot()`, and `truncate()` sequence to add data to the log.

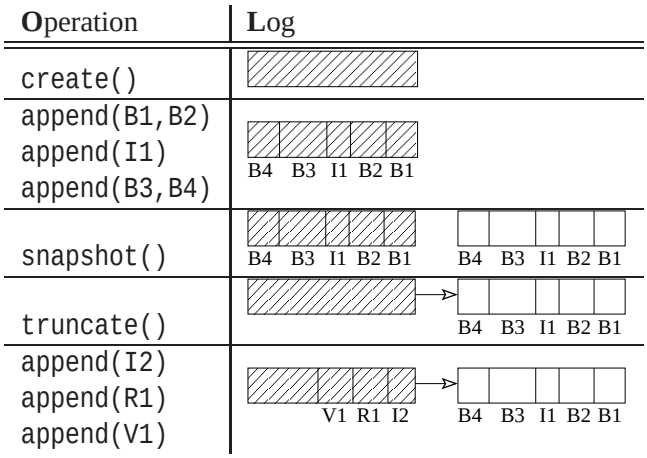


Figure 7.2: This example illustrates how the client library uses the extended API to write the first version of the file system shown in Figure 6.4. The shaded extent is the mutable log head; immutable extents are shown in white.

7.2.2 Reading data from a Secure Log

To provide random access to any element in the log, the system implements *two-level naming*. In two-level naming, each block is addressed not by a single name, but by a tuple. The first element of the tuple identifies the enclosing extent; the second element names the block with the extent. Retrieving data from the system is a two-step process. The system first locates the enclosing extent; then, it extracts individual application-level blocks from the extent. Two-level naming reduces the management overhead incurred by the infrastructure by decoupling the infrastructure's unit of management from the client's unit of access. The infrastructure needs only to track data at the extent level, and the client can still address individual blocks. Both blocks and extents are self-verifying.

When an application writes a block to the log, the block is stored in the mutable extent at the head of the log. Because the log head is a mutable extent, the system can not know the name of the hash-verified extent where the block would eventually and permanently reside. To resolve this problem, each extent is assigned an integer corresponding to its position in the chain. When data is appended to the log, the address returned to the application identifies the enclosing extent by this counter. Each extent records the mapping between counter and permanent, hash-verified extent name for the previous extent.

To read data embedded in extents, the client must know the extent name, location of server storing extent, and block name. However, the application only records the extent counter

and block name since it does not know the extent name at the time of `append()`. As a result, the client library must first resolve the extent counter to extent name. In particular, the client library first accesses the mappings stored in the log head via `get_map()` to determine the name of the previous extent. We assume that a mechanism exists to locate the log head or any extent given extent name (Chapter 9). Next, the client continues retrieving and resolving extent mappings until it locates the extent holding the data. It then uses the `get_blocks()` operation to retrieve the requested blocks from that extent. To accelerate the translation between counter and extent name, the client library caches the mappings. Also, in implementation, each extent contains not just the mapping for the previous extent, but a set of mappings that allow resolution in a logarithmic number of lookups.

7.2.3 Other benefits of Aggregation

Aggregating blocks into extents and extents into a log improves the system's efficiency in several ways. First, breaking a log into extents enables servers to intelligently allocate space for extents that may have a maximum size while the log itself can grow to be arbitrarily large. Second, extents decouple the infrastructure's unit of management from the client's unit of access. As a result, the storage infrastructure can amortize management costs over larger collections of data. Third, two-level naming reduces the query load on the system because clients need to query the infrastructure only once per extent, not once per block. Assuming data locality—that clients tend to access multiple blocks from an extent—systems can exploit the use of connections to manage congestion in the network better. Finally, clients writing multiple blocks to the log at the same time need only to create and sign a single certificate.

7.3 A Distributed Secure Log and Error Handling

A client library should communicate the result of an operation to applications. However, determining the return value can be difficult since a secure log is replicated and replicas are distributed throughout the wide-area for durability. As a result of maintaining consistency across the distributed log replicas, there are three possible return values: sound, unsound, and undefined. A sound result is durable meaning it will persist despite server failure. An unsound result is a failure and will not be maintained by the system. Finally, an undefined result has unknown status. An undefined result is often not returned to the application. Instead the latest replicated state is obtained via calling `get_cert()` on the servers. If the less than a threshold of servers respond or the replicated

state is in an inconsistent state (e.g. concurrent write attempts), the client triggers a repair audit. If the repair audit repairs the log to the latest sound write, and the request can be applied from that write, then the request is resubmitted. This process of retrieving the latest replicated state, triggering a repair audit, and resubmitting the request continues until the request is either sound or unsound. In particular, `append()` and `truncate()` operations follow this process until the request is either sound or unsound. As a result, we do not discuss `append()` and `truncate()` further. `create()`, `snapshot()`, and `put()`, however, requires an extra step that first interacts with the administrator.

The `create()`, `snapshot()`, and `put()` operations create new extents and communicate with the administrator. `create()` creates a mutable log head. `snapshot()` and `put()` create immutable hash-verified extents. Each of these operations require a set of servers called a configuration to be allocated to host the new extent. When the administrator creates a configuration, it has three possible responses: success, failure, or unknown, or success. If the administrator returns success then a configuration signed by the administrator is returned to the client. The signed configuration contains the servers responsible for hosting the new extent. The administrator returns failure in two situations. If no storage servers are allocated (e.g. perhaps the client has used its quota). Also, if the extent already exists and has been repaired at least once. In either case, the error would be returned to the client. If no response is received from the administrator possibly due to a network transmission error, then the result is unknown. The request can be resent until the request succeeds or fails; each request is idempotent.

The client sends the request (`create()`, `snapshot()`, or `put()`) along with the signed configuration to the new storage servers. The result of this operation can be sound, unsound, or undefined similar `append()` and `truncate()`. If the result is sound (a threshold of servers responded with success), then the new extent is durable and success is returned to the application. If however, the request is unsound, then failure is returned to the application. A result is unsound if the extent has existed for a while such that data has already been appended (perhaps another client) or the extent has already been repaired at least once. Finally, if the request is undefined, the request is continually resubmitted, latest state obtained (via `get_cert()`), and repair audit triggered.

7.4 Discussion

There are five reasons why implementing a secure log interface is easier to implement in a secure way than other interfaces. Most reasons are related to the structure of the secure log itself.

First, since most of the log is immutable and stored in hash-verified extents, the order and

data integrity of those extent replicas are immediate – the extent name verifies both the order and content of a hash-verified extent. It is not possible for a server to corrupt a hash-verified extent.

Second, the order and data integrity of the head of the log can be verified using the verifier contained in the certificate. This verifier ensures the order and data integrity of the entire log. There is only one sequence of appends that results in a particular verifier.

Third, the verifier provides a "natural" predicate that can be used to ensure the consistency of a log. Each storage server checks that the predicate verifier matches local state before applying any operation. Furthermore, the verifier contained in a new certificate can be used to ensure the integrity of the subsequent state. The verifier is cheap to compute and update.

Fourth, the narrow interface to modify a log reduces the complexity of the handling errors. Sound and unsound results return success or failure to the application. Undefined results require more decision processing. The client library needs to decide to resubmit the request, obtain the latest state of the log, or trigger a repair audit. Fortunately, the methods that modify the log are few and the ways in which the log can be modified are even fewer.

Finally, a secure log decouples the infrastructure's unit of management (extent) from the client's unit of access (data block). As a result, the storage infrastructure can amortize management costs over larger collections of data.

Chapter 8

Dynamic Byzantine Quorums for Consistency and Durability

The last chapter illustrated that the implementation of a secure log interface must account for replicating the log. In particular, a design for replicated, log-based storage must ensure consistency of the log head and durability of all log elements in the presence of a variety of server and network failures. Typical server failures that we may hope to tolerate include transient failure such as reboot, permanent failure such as disk failure, and erroneous failure such as database corruption and machine compromise. Furthermore, we hope to tolerate network failure that may include loss of network connectivity such as temporary partition and transmission failure such as message drop, reorder, delay, or corruption.

Our approach relies on dynamic Byzantine quorums. In general, Byzantine quorum protocols tolerate many server and network failures and maintain consistency over replicated state. Basic Byzantine quorum protocols tolerate a threshold of faulty servers in a configuration, set of storage servers that maintain replicated state; however, configurations are static and not allowed to change. In particular, the level of protection against failure degrades overtime as faults accumulate. This is because basic Byzantine quorum protocols can only tolerate a finite number of failures over the lifetime of the system. Dynamic Byzantine quorum protocols, on the other hand, extend basic quorum protocols and allow for reconfiguration; they allow the set of servers responsible for storing replicated state to change. As a result, dynamic Byzantine quorums can tolerate many failures overtime (assuming a limited number of failures within a specific window of time).

In this chapter, we demonstrate how we use dynamic Byzantine quorums to write data

Configuration contents:	
object_id	cryptographically secure name of object
client_id	hash of client's public key: H(PK)
ss_set[]	set of storage servers: set of H(ss_PK)
f	fault servers tolerated
seq_num	configuration sequence number
timestamp	creation time of configuration
ttl	time the configuration remains valid

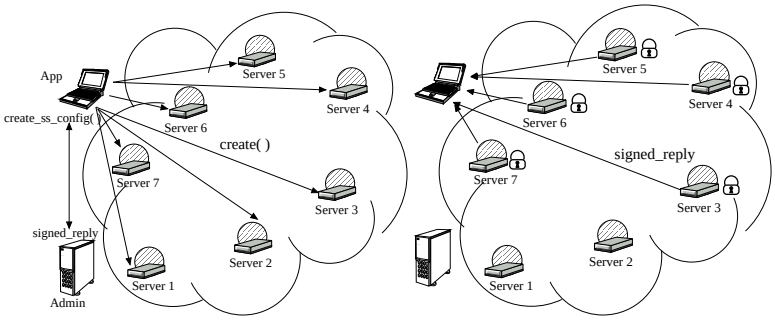
Table 8.1: A configuration defines a set of storage servers that maintain a replicated log.

to a log in a consistent manner and to ensure the durability of all log elements. The system can make progress, add new data to the log, with up to f failures in a configuration. It can also create a new configuration to assume responsibility for storing a portion of the log with up to $2f$ failures in the old configuration. This construction is an improvement over previous constructions which only allow up to f failures in a particular configuration.

The rest of this chapter describes the protocol requirements, assumptions, details, and correctness. In Section 8.1, we discuss background and prior work. Next, in Section 8.2, we discuss the protocol requirements. We list our assumptions in Section 8.3. We describe the quorum repair requirements in Section 8.4. In Section 8.5, we present the protocol details. We show that the protocol satisfies the requirements in Section 8.6. Finally, we discuss how a secure log makes implementing the protocol easier in Section 8.7.

8.1 Background and Prior Work

Byzantine fault-tolerant quorum protocols can ensure consistency of replicated state. A quorum protocol is executed over a *configuration*, set of storage servers that maintain replicated state. The parameters that define a configuration are shown in Table 8.1. To update the replicated state, a *quorum* of servers in a configuration must agree to the change. A quorum is a threshold of servers and its size is defined by the number of servers in a configuration n and number of faulty servers f the protocol should tolerate[MR97]. For example, a quorum might have $q = n - f$ servers where $n > 3f$ and can tolerate f faulty servers. Figure 8.1 shows a client attempting to create a log with a configuration that includes seven servers and can tolerate two faulty servers ($f = 2$ and $n = 7 > 3f$). After an administrator selects a configuration, the client submits the `create()` request to all the servers in the configuration. The `create()` request succeeds after the client receives



(a) `create()` request (b) `create()` acknowledgment

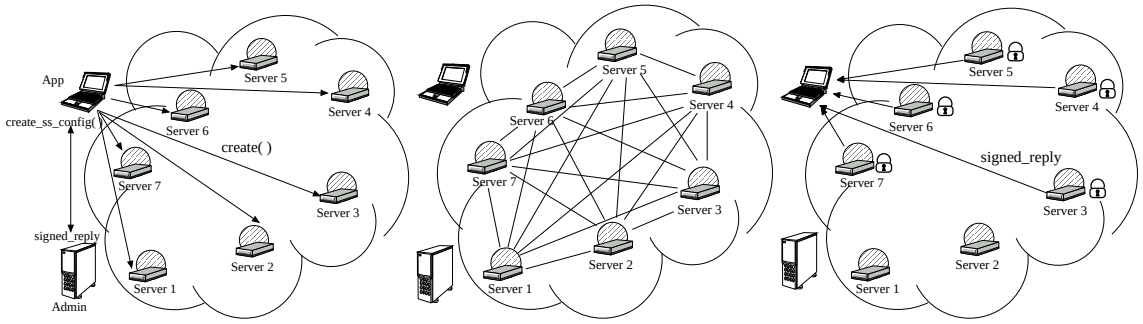
Figure 8.1: Example `create()` request using a Byzantine fault-tolerant quorum. (a) A client attempts to create a secure log with a configuration that includes seven servers and can tolerate two faulty servers ($f = 2$ and $n = 7 > 3f$). After an administrator selects a configuration, the client submits the `create()` request to all the servers in the configuration. (b) The `create()` request succeeds after the client receives positive acknowledgment from a quorum of servers ($q = 7 - 2 = 5$).

positive acknowledgment from a quorum of servers ($q = 7 - 2 = 5$).

Byzantine fault-tolerant agreement protocols similarly maintain consistency of replicated state; however, they do so at a higher communication cost than Byzantine quorum protocols [AGG+05]. In the common case, Byzantine agreement protocols use $O(n^2)$ messages over multiple rounds, whereas Byzantine quorums use $O(n)$ over two rounds where the second round is often piggybacked onto subsequent operations [AGG+05]. Figure 8.2 shows a client creating a log using a Byzantine agreement protocol.

Both Byzantine agreement and quorum protocols tolerate up to f faulty servers and ensure consistency. However, traditional Byzantine fault-tolerant quorum (and agreement) protocols do not allow changes to a configuration. They guarantee correctness only if no more than f servers in a configuration fail *during the life of the system*. This restriction is impractical for long-running systems. Such systems need the ability to change the configuration over time. They must be able to remove faulty servers from the configuration, replacing them with new servers. Some systems may even wish to change the size, and thus fault-tolerance, of a configuration. *Dynamic* Byzantine quorum protocols allow the system to change a configuration by performing a *repair* operation.

Martin and Alvisi [MA04] first defined a framework for dynamic Byzantine quorum protocols that could maintain consistency across multiple configurations. In that work, they identify two properties, *soundness* and *timeliness*, that when satisfied, guarantee consistency in a dynamic environment. Informally, soundness ensures data read by a client was previously written to a quorum of servers; timeliness, on the other hand, ensures the data read is the most recent value written.



(a) `create()` request (b) multiple round agreement (c) `create()` acknowledgment

Figure 8.2: Example `create()` request using a Byzantine fault-tolerant agreement. (a) and (c) are similar to the Byzantine quorum `create()` request and acknowledgment in Figure 8.1.(a) and (b), respectively. However, Figure (b) above illustrates that Byzantine agreement protocols use $O(n^2)$ messages over multiple rounds, whereas Byzantine quorums use $O(n)$ over two rounds where the second round is often piggybacked onto subsequent operations [AGG+05].

They call these properties *transquorum* properties because they do not depend on quorum intersection between configurations [MR97]. Finally, they prove that transquorum properties are sufficient to guarantee the consistency semantics provided by each of the protocols they consider.

The Martin and Alvisi protocol can invoke a repair protocol with up to f faulty servers in a configuration; however, in implementation there is a non-zero probability that more than f failures can occur. In implementation, repair cannot be invoked when it is needed most, when less than a quorum of servers are available. Essentially, a quorum of servers in a configuration are required to agree to trigger a repair protocol that will create a new configuration. Assuming f servers are always faulty in a configuration, such a protocol often periodically triggers repair, but cannot "react" to failures when necessary.

In this chapter, we extend the Martin and Alvisi dynamic Byzantine quorum protocol to allow the servers to reactively invoke a repair protocol when less than a quorum is available.

8.2 Protocol Requirements

In this section, we outline the requirements of a dynamic Byzantine quorum protocol that can invoke a repair protocol with less than a quorum of server responses in a configuration. The requirements are consistency, durability, and liveness.

- *Consistency* means all successful writes can be totally ordered.

- *Durability* means no successful writes are lost.
- *Liveness* means new write attempts can eventually succeed; also, known as *progress*.

A write attempt is a client issued operation to all the servers in a configuration requesting each server modify its local state. In particular, clients submit write attempts and servers can either accept or reject the request (they can also ignore requests). A server that accepts the request invokes the write operation on its local replica and replies accordingly. A client's write attempt is successful after receiving sufficient server accept responses. We discuss the base write protocol in Section 8.5.1.

8.3 Protocol Assumptions

We limit our assumptions to ensure consistency and durability of the secure log under the broadest possible conditions. First, we assume clients perform operations on extents of the log as defined in Table 7.2. Each extent has its own configuration defined and signed by an administrator. The configuration includes the public key of each storage server which can be used to verify a server's response. Additionally, server public keys can be used to set up authenticated point-to-point channels. Second, we assume servers are computationally bound so that cryptographic primitives are effective. Further, we assume that servers have persistent storage that is durable through a crash and subsequent recovery (transient failure). Finally, we assume an asynchronous timing model; no assumptions are made about the duration of message transmission delays. Channels can drop, reorder, and duplicate messages; however, channels are assumed to be fair, a message sent an infinite number of times will be received an infinite number of times [ACT00].

We assume clients are correct, follow specification, except for crashing and recovering. Furthermore, we assume the administrator is correct and always available. We discuss our assumptions for the number and types of faulty servers tolerated below.

Server Fault and Attack Model

We assume that each server is either correct or faulty. A *correct* server follows its specifications. A Byzantine *faulty* server, however, can deviate from its specification. Since Byzantine failure is a generalization of all failures, we use a hybrid server fault model that breaks Byzantine servers into three types of faulty servers: benign, unwitting stale, and malevolent (correct and up-to-date, correct and out-of-date, and malicious, respectively).

A *benign* server is correct and follows its specification except when suffering from transient or permanent failure: crashing and (potentially) recovering. We assume a benign server exe-

cutes a transient failure protocol that brings its state up-to-date after transient failure. The transient failure protocol is discussed in Section 8.5.5. An *unwitting stale* server is correct but unaware that its state is out-of-date. Possibly due to a network error such as a dropped message or a network partition that caused the server to unknowingly miss some updates. As soon as an unwitting stale server realizes its state is out-of-date (e.g. receiving a request with a later sequence number), it performs a transient failure protocol to update its local state. A *malevolent* server exhibits out-of-specification, malicious, non-crash behavior and may attempt to subvert protocols.

The problem is that a client cannot differentiate a malevolent server response from a benign or unwitting stale server. A non-responsive malevolent server has the same symptoms as a benign server that is unavailable due to a transient or permanent failure; no response is received in both cases. Similarly, a malevolent server can have the same symptoms as an unwitting stale server; both can reply with out-of-date information whether intentional or unintentional. We consider both symptoms, non-responsiveness and stale data, server attacks. Fortunately, malevolent servers are restricted to these two attacks since malevolent servers cannot undetectably alter self-verifying data. As a result, the protocol should be designed to tolerate both attacks.

We assume at most $2f$ servers are faulty in a particular configuration with at most f malevolent servers.

8.4 Quorum Repair

In this section, we discuss the challenges for a quorum repair protocol. Quorum repair is necessary to satisfy the durability requirement since we assume that eventually all servers permanently fail. It transfers state to new servers during reconfiguration while maintaining consistency. We use a quorum repair protocol adapted to our target operating environment. Specifically, the algorithm ensures no successful writes are lost during reconfiguration with up to $2f$ faulty servers in the old configuration where at most f of those servers are malevolent. In particular, the quorum repair protocol maintains the latest successful write across configurations.

8.4.1 Challenges

Repair is a protocol that creates a new configuration for a particular object and initializes the new configuration to the latest successful write value of the old configuration. It is needed to remove faulty servers, add new servers, change configuration parameters, and ensure new write

Time	Config	Servers								
		1	2	3	4	5	6	7	8	9
1	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
2	0	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*b*	*b*
3	0	*c*	*c*	*c*	*c*	*c*	X	X	*b*	*b*
Reconfiguration		1	2	3	4	5	6’	7’	8	9
4	1	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*

(a) *c* is last successful write before server failure

Time	Config	Servers								
		1	2	3	4	5	6	7	8	9
1	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
2	0	*c*	*c*	*c*	*c*	*c*	*c*	*b*	*b*	*b*
3	0	*c*	*c*	*c*	*c*	*c*	X	X	*b*	*b*
Reconfiguration		1	2	3	4	5	6’	7’	8	9
4	1	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*

(b) *a* is last successful write before server failure

Figure 8.3: A write is successful after a client receives positive acknowledgment from a quorum of q storage servers. Two clients simultaneously submit conflicting writes. During repair, the system should initialize the new configuration to the state reflecting the latest successful write. In these two examples, the server state that can be observed from the clients at time $t = 3$ is the same, but the latest successful write differs. In (a), the client that wrote *c* received a quorum of positive server acknowledgments and, thus, is successful. In (b), the client that wrote *c* did not receive a quorum of positive server acknowledgments so the write failed, thus, the new configuration must be initialized to *a*.

attempts can eventually succeed. The goal of repair is to ensure consistency, durability, and liveness of a particular object across multiple configurations.

When selecting a quorum repair protocol for use in the wide-area, we must carefully consider several issues. First, we must consider when to trigger repair. Because some failures, such as power failures or network partitions, may knock out groups of servers, it is important that the algorithm be able to tolerate large numbers of failures. We must also ensure that the conditions for triggering repair do not allow a small group of malevolent servers to initiate needless repair.

During repair a new configuration is chosen to host the replicated log. The new configuration must be initialized with the correct state of the log. It is critical that no successful writes are lost and no failed writes are elevated to success status.

Consider the examples in Figure 8.3 to understand the challenges in initializing the state of a new configuration, it shows the client’s view of the storage system. The log is replicated on nine storage servers, can tolerate two faulty servers, and requires seven positive server acknowledgments

for a successful write ($n = 9$, $f = 2$, and $q = 7$, respectively. $n = 4f + 1$). Negative server acknowledgments contain the latest write accepted by a server. Assume two clients, a workstation and a laptop, simultaneously submit conflicting write attempts. Figure 8.3(a), at time $t = 2$, shows the workstation attempting to write b, predicated on a, to the log head. The workstation receives only two positive server acknowledgments (servers 8 and 9) and seven negative (servers $1-7$), so the request fails. At the same time, the laptop attempts to write c, predicated on a. The laptop receives a quorum of positive server acknowledgments ($1-7$), and thus is successful. At time $t = 3$, servers 6 and 7 permanently fail. During repair, the system should initialize the new configuration to state c. But, the configuration does not contain a quorum of responsive servers to confirm that c was indeed successful.

Compare this with the example shown in Figure 8.3(b) where both client write attempts fail. Again, at time $t = 2$, the workstation attempts to write b predicated on a, but receives positive acknowledgments from only three servers ($7-9$) and negative acknowledgments from the rest (servers $1-6$), so the request fails. At the same time, laptop attempts to write c predicated on a which also fails. It receives positive acknowledgments from six servers ($1-6$) and negative acknowledgments from the rest (servers $7-9$). In this example, both requests fail. When servers 6 and 7 permanently fail at time $t = 3$, the latest successful write is still a. Comparing examples 8.3(a) and 8.3(b), the state visible from the available servers at time $t = 3$ is the same in both examples; however, the repair algorithm needs to initialize the new configuration to different values.

Note that in the above example, after one round of the client submitting a write request and receiving responses, only the client knows that a write is successful. In the base write protocol discussed in Section 8.5.1, however, there is a second round where the client sends write success confirmation to the servers. The client classifies a write as successful only after the second round completes, after receiving a quorum of servers responding to the confirmation. We assume that the client library only reports success to the application after completing the second round.

If, on the other hand, the client receives sufficient negative acknowledgments, then it reports failure to the application. Alternatively, if the client library does not receive sufficient response in either round, it times out. After timing out, the client library performs a quorum read. If the replicated state changed (e.g. servers store another write), then the client library returns failure to the application. If the quorum read fails (times out), then the client triggers a repair audit. The client library repeats the sequence of sending write attempts, quorum reads, and triggering repair audits until it is certain the write attempt succeeds or fails on a sufficient number of servers (e.g. $q = n - f$ positive or negative acknowledgments for success or failure, respectively). Section 7.3 describes

this sequence as well. In any case, the client library only reports success to the application after the second round completes.

8.4.2 Triggering repair

It is not appropriate to assume that the client will always be on-line to monitor configurations and trigger repair. Instead, the storage servers in the configuration must initiate repair when it is required. We use a *reactive* approach to triggering repair. A storage server requests repair in two situations: when it believes between $f+1$ to $2f$ servers have failed or the replicated state is *wedged* – cannot make progress because replicas are in an inconsistent state. At least $2f+1$ storage servers must agree that repair is needed before reconfiguration is initiated.

$2f+1$ is the minimum number of servers required to trigger repair. If only f servers were needed to trigger repair, then f malevolent servers could waste system resources by triggering repair continuously. Alternatively, if storage servers requested repair after only f servers failures, then f malevolent servers could force repair continuously simply by being unresponsive. Furthermore, a protocol could not guarantee that the latest successful write will be used in a repair with $2f$ or less servers requesting repair. In particular, there is no way to guarantee that f malevolent and f unwitting stale servers are not used to initiate repair using an old value. Theorem 1 states this observation.

Theorem 1. *Given f servers may be malevolent and f correct servers may be out-of-date, $2f+1$ is the minimum number of servers required to trigger repair with the latest successful write.*

Proof. Proof by contradiction. Assume that $2f$ servers is sufficient to trigger repair with the latest successful write. Then in all cases, at least one out of $2f$ server repair requests contains the latest successful write. However, this is not possible for all cases. In particular, $2f$ server responses may include f malevolent servers responding with old values and another f servers that may be out-of-date (f correct servers may be out-of-date since quorum protocols allow progress without response from f servers). In the other direction, we again use proof by contradiction. Assume that $2f+1$ servers is not sufficient to trigger repair with the latest value. Then all $2f+1$ repair requests contain out-of-date information. However at most f servers are malevolent and at most f correct servers are not involved in the latest successful write. As a result, this is not possible since at least one correct server must be involved in the latest successful write, thus at least one server is correct and up-to-date. □

Soundness proof contents:	
cert	certificate
config	configuration
ss_sigs[]	$2f+1$ or more signatures
	$<$H(cert+config)$>_{ss_priv}$

Table 8.2: A soundness proof can be presented by any machine to any other machine in the network to prove that a write was sound. To provide this guarantee, the proof contains a set of q storage server signatures over an append's certificate (Table 6.1) and the storage configuration (Table 8.1).

Operation	Soundness Proof {cert_seq_num, config_seq_num} – sigs	Configuration Servers
`create()`	{0, 0} – sigs{1, 2, 3, 4, 5, 6, 7}	{1, 2, 3, 4, 5, 6, 7, 8, 9}
`append()`	{1, 0} – sigs{1, 2, 3, 4, 5, 6, 7}	{1, 2, 3, 4, 5, 6, 7, 8, 9}
`repair()`	{1, 1} – sigs{1, 2, 3, 6', 7', 8, 9}	{1, 2, 3, 4, 5, 6', 7', 8, 9}
`append()`	{2, 1} – sigs{1, 2, 3, 6', 7', 8, 9}	{1, 2, 3, 4, 5, 6', 7', 8, 9}
`append()`	{3, 1} – sigs{1, 2, 3, 6', 7', 8, 9}	{1, 2, 3, 4, 5, 6', 7', 8, 9}
`repair()`	{4, 2} – sigs{1, 4', 5', 6', 7', 8, 9}	{1, 2, 3, 4', 5', 6', 7', 8', 9'}
`truncate()`	{5, 2} – sigs{1, 4', 5', 6', 7', 8', 9'}	{1, 2, 3, 4', 5', 6', 7', 8', 9'}
`repair()`	{5, 3} – sigs{3', 4', 5', 6', 7', 8', 9'}	{1', 2', 3', 4', 5', 6', 7', 8', 9'}
`append()`	{6, 3} – sigs{3', 4', 5', 6', 7', 8', 9'}	{1', 2', 3', 4', 5', 6', 7', 8', 9'}

Figure 8.4: Example total order of sound operations.

Instead of triggering repair reactively, other systems trigger repair in a proactive manner at regular intervals [CL00]. Such systems require a quorum to trigger repair. Generally, these systems are not able to trigger repair when it is needed most, when more than f servers have failed in the repair interval. Reactive repair can be used in combination with proactive repair to supplement this deficiency adding the ability to maintain consistency, durability, and liveness when more than f servers fail. The key difference is a strictly proactive repair system requires at least $n = 3f+1$ servers [MA04]; where as, a reactive repair system requires at least $n = 4f+1$ servers since the system reacts after $f+1$ or more failures have been detected. Both proactive and reactive repair systems require at least $r = 2f+1$ servers to trigger repair (Theorem 1).

8.4.3 Initializing a configuration after repair

To ensure that no successful writes are lost during repair, we base repair on a *soundness proof*. Table 8.2 shows the contents of a soundness proof. It includes a certificate (Table 6.1), configuration (Table 8.1), and a quorum q of server signatures over a hash of the certificate and

Time	Config	Servers								
		1	2	3	4	5	6	7	8	9
3	0	*a*/c	*a*/c	?	?	*c*	?	?	*a*	*a*

Figure 8.5: Latest soundness proof. From Figure 8.3(a), at time $t = 2$, the latest sound write was *c*. Assume a quorum of servers (servers 1-7) acknowledged receiving the latest soundness proof (configuration parameters $n = 9$, $q = 7$, $r = 5$, and $f = 2$). This figure shows the administrator's view of the storage system at time $t = 3$ after receiving replies from five servers (servers 1, 2, 5, 8, 9). Assume servers 1 and 2 are malevolent and can either send the latest or old proof and servers 8 and 9 are out-of-date and did not receive the last proof. At least one server response out of five (server 5) contains the latest soundness proof (*c*).

configuration. It can be stored by and presented to any server as proof that a write was successful – *sound*.

A soundness proof contains enough information to determine the latest sound write since both the certificate and configuration contain a sequence number which defines a total ordering of sound writes. Figure 8.4 shows an example total ordering of all operations on the head of a log.

The use of soundness proofs also enables repair with up to $2f$ faulty servers in a configuration while maintaining data consistency. By collecting the most recent soundness proofs from $2f + 1$ servers in a configuration, we can be sure that we have retrieved a soundness proof from at least one non-faulty server that participated in the last sound write (Theorem 1). Figure 8.5 illustrates that $2f + 1$ server responses contains the latest soundness proof from at least one correct and up-to-date server.

Other systems use the notion of *repairable* writes to initialize new configurations [AGG+05]. Repairable writes are *not* successful writes, rather replies from only $f + 1$ servers (not the required number of quorum replies). We did not use this solution since $f + 1$ responses is not sufficient to ensure that an old value is not used instead of the latest.

8.4.4 Certificates and Soundness Proofs

Table 8.2 shows the contents of a soundness proof (which includes the contents of a certificate and configuration, Table 6.1 and Table 8.1, respectively). The client increments the certificate sequence number and signs the new certificate before invoking each operation. To create a soundness proof, the client must receive a quorum of responses from the storage servers, where each response is a signature over the secure hash of the new certificate and configuration. The set of all soundness proofs defines a total ordering of all sound writes since it contains a unique pair of sequence numbers from the certificate and configuration. This total order is sufficient to maintain

consistency [Sch90]. Furthermore, soundness proofs are used to maintain the durability of sound writes. Section 8.6 proves the correctness.

8.4.5 The Repair Audit Protocol

Though only the storage servers can trigger repair, any component of the system can request a repair *audit* (e.g. client, storage server, administrator, periodic timer, etc.). When a storage server receives a repair audit, it attempts to read the latest state (certificate and soundness proof) from the other servers in the configuration. If a quorum responds and the data is in a consistent state, the storage server takes no action. If, however, a quorum does not respond or the replicas are in an inconsistent state (wedged), then the storage server will create a repair request, record it in local stable storage, and submit it to the administrator. If the server observes that it already stores a signed repair request on its local disk, it will forward the same request to the administrator. Once a storage server is in the repair state, it does not accept updates until a new configuration is created.

A minimum of $2f+1$ repair requests are required to trigger the repair protocol. This ensures the latest sound value will be used in the repair (Theorem 1). Additionally, $2f+1$ servers in the repair state ensures no more writes will succeed in the current configuration [MA04]. If at most f servers are malevolent, then at least $f+1$ of the $2f+1$ required server requests to trigger repair are correct servers. As a result, at least $f+1$ correct servers will not accept new writes ensuring progress cannot be made in the current configuration.

8.5 Protocol Details

In this section, we describe how components of the system interact to implement the `create()`, `append()`, and `repair()` protocols (Table 7.2). In the basic system there are three components: the client, the storage servers, and an administrator. We require that all updates submitted to each storage server include a certificate signed by a private key that is known only to the client. The storage servers are replicated state machines that store data, transform local state with well defined procedures, and cryptographically sign acknowledgments testifying to that fact. Finally, an administrator is required to create configurations. The administrator is the authority on the latest configuration for an object. A storage server will participate in the management of an object when it observes that it is included in a configuration signed by the administrator. We assume that the administrator is non-faulty.

Field	Description
proof	Certificate, configuration, and a quorum of server signatures.
block_names[ ]	Secure hash of data blocks (block_names[i]=H(data_blocks[i])).
data_blocks[ ]	Data.
mapping	Previous extent_counter and extent_name.
pending_proof	Certificate, configuration, and no server signatures.
pending_block_names[ ]	Secure hash of pending data blocks.
pending_data_blocks[ ]	Pending data.
pending_map	Pending previous extent_counter and extent_name.
pending_operation	`create()`, `append()`, `truncate()`, `snapshot()`, `put()`

Figure 8.6: Local server state for log head and hash-verified extents. It includes proven state (with soundness proof) and pending state (without soundness proof). Proven state includes the latest soundness proof, block_names, and data. Mapping is used to connect extents into a secure log. Proven state is null when an extent is first created, when `create()`, `snapshot()`, or `put()` are pending; otherwise, it is not null. Pending data includes a pending soundness proof (certificate and configuration without server signatures), block_names and data. Pending_map is used by `truncate()`, pending_map points to the extent created during `snapshot()`. Pending state is null if no requests are pending. When a pending request gathers proof of soundness the pending_proof field replaces the proof. `create()`, `snapshot()`, and `put()` replace block_names, data_blocks, and mapping with the associated pending fields. `append()` adds the pending_block_names and pending_data_blocks to block_names and data_blocks fields, respectively. `truncate()`, however, removes block_names and data_blocks fields; additionally, it replaces the mapping field with the pending_map field.

8.5.1 Base Write Protocol

The base write protocol works as follows. There are two rounds; however, the second round is often sent with a subsequent operation. The client library does not report success to the application until the second round completes successfully. First, a client submits a request to the storage servers. The request includes a predicate verifier, new certificate that contains a new verifier and new sequence number, and an associated client signature. When a storage server receives the message, it checks the request against its local state. If the request satisfies all conditions, the server stores the data to non-volatile storage and responds to the client with a signed positive acknowledgment. The client combines signed positive acknowledgments from a quorum of servers to create a soundness proof (quorum size $q = 3f + 1$ and configuration size $n = 4f + 1$). Next, in the second round, the client sends the soundness proof to the servers, often as part of a subsequent operation. Each server stores the soundness proof to a stable storage and responds to the client. The client can be certain the log has been written successfully after sending the soundness proof to all servers and receiving responses from a quorum of servers. Section 8.6 proves the correctness of this protocol.

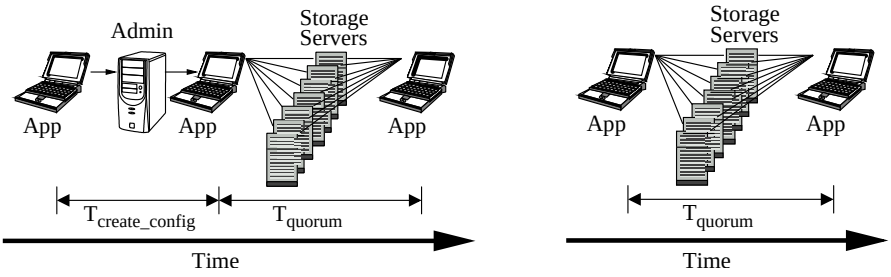


Figure 8.7: (a) To complete a `create()` request, a client must first request a new configuration from the administrator. The client then sends the configuration along with the signed certificate to storage servers listed in the configuration. (b) To complete an `append()` request, the client must only send a message to each storage server in the configuration.

Figure 8.6 shows the local state stored by each server of the log head or hash-verified extent. The pending fields are null if there are no pending requests. If the pending fields are not null and remain unchanged for a specified amount of time, the server will query a quorum of other storage servers in its configuration to gather the signatures required to construct a soundness proof. When a server's pending state stores sufficient signatures for a soundness proof, the server updates the proven state with the pending state. It replaces the proof field with the pending_proof field. The `create()`, `snapshot()`, and `put()` operations replace the block_names, data_blocks, and mapping with the associated pending fields. `append()` appends the pending_block_names and pending_data_blocks to block_names and data_blocks fields, respectively. `truncate()` removes block_names and data_blocks fields; additionally, it replaces the mapping field with the pending_map field. Notice that only state that a server can locally prove is sent to the administrator in a repair request.

8.5.2 The `create()` Protocol

The `create()` protocol is used to create a new log by allocating space on servers in a configuration. The protocol is illustrated in Figure 8.7(a). Note that the `snapshot()` and `put()` protocols are nearly identical to the `create()`. The difference is the type of extent created and whether data blocks are included in the request. `create()` creates a mutable log head, `snapshot()` and `put()` create immutable (read-only) hash-verified extents. Furthermore, `create()` does not include any data blocks since the log head is initially empty. Whereas, `put()` contains data blocks in the request and `snapshot()` instructs servers in the new snapshot configuration to download the

data blocks from the servers in the log head configuration. Despite the differences between the type of extent created and inclusion of data blocks in the request, the protocol between the client, administrator, and servers are similar. As a result, we only discuss the `create()` protocol in detail.

To begin the `create()` protocol, the client submits a request to the administrator to create a new storage configuration. This request includes a certificate signed by the client that allows the administrator to verify that the client is permitted to create a log. The administrator then chooses a set of servers to manage the log and returns the configuration, signed by the administrator's private key, to the client. Next, the client sends the configuration and its original create certificate to all of the storage servers listed in the configuration. When a storage server receives the message, it allocates space for the log and stores the certificate and configuration to non-volatile storage. After the data reaches stable storage, the storage server responds to the client with a signature over the secure hash of the certificate and configuration. After collecting $q = 3f + 1$ signatures, the client combines the signatures to create a soundness proof for the `create()` operation.

At this point, the client knows the `create()` is sound, but the servers do not. As a result, the client sends the soundness proof to all the servers and waits for acknowledgment. When the servers receive the soundness proof, they record it to stable storage and respond to the client. A server that is not expecting the soundness proof (e.g. missed the initial `create()` request due to network transmission error), stores the proof to stable storage and responds to the client[1]. After receiving acknowledgment from a quorum of servers, the client can be certain the log has been created and will remain durable.

The client library reports success to the application after this second round completes. The second round is often performed with subsequent operations. For instance, `append()`, `truncate()` or any of the `get()` operations may include this soundness proof.

8.5.3 The `append()` Protocol

The `append()` operation adds data to the log, moving the replicated state from one consistent, *safe* state to another. The protocol is illustrated in Figure 8.7(b). Note that the `truncate()` protocol is equivalent to the `append()` except the `truncate()` removes data blocks stored at the log head and updates the mapping (previous extent counter to extent name mapping); whereas, the `append()` adds more data blocks to the log head. As a result, we only discuss the `append()` protocol in detail.

[1] A certificate and configuration is the only state associated with a `create()`. As a result, a `create()` soundness proof contains enough state to update a server that missed the initial `create()` request.

To begin the `append()` protocol, the client creates and signs a certificate that describes the changes to the log. The request describes the current state of the log and the next state of the log, should the `append()` operation succeed. The certificate also includes a sequence number that must be greater than the number in the log's current certificate. The client then sends the request—which includes the data to be added, the certificate, and the previous soundness proof—to each storage server.

Each storage server determines if the request can be applied. Success of the request is predicated on several checks. The certificate must include a valid signature and sequence number. The previous soundness proof contained in the request must match the current state recorded by the storage server. If the conditions are met, the storage server writes the new data to the log on its local store. It then responds to the client with a signature over the secure hash of the new certificate and the current configuration. After collecting $q = 3f + 1$ signatures, the client combines the signatures to create a soundness proof for the operation.

With this protocol, a storage server receives a soundness proof in the second round or in a subsequent operation. If the client stops submitting operations or does not send the soundness proof by itself, however, a storage server is left with a soundness proof that is stale by one operation. To obtain a current soundness proof, if a storage server does not receive the soundness proof in a specified amount of time, it will query a quorum of storage servers in its configuration to gather the signatures required to construct a soundness proof. It is important that storage servers store the latest soundness proof since they use soundness proofs during repair. In Part IV, we will describe the additional measures taken in the implementation to ensure that a current soundness proof is available to the storage servers. Essentially, the implementation of the system stores the soundness proof in a distributed hash table (DHT) before responding to the client. Storage servers check the DHT for soundness proofs before triggering repair.

If the client does not receive a sufficient number of responses, the client cannot be sure that the write is sound and durable. As a result, the client library does not report success until it receives acknowledgment from a quorum of servers after sending them the associated soundness proof. The client library may be able to commit data to the log by retrying the operation. If the problem is transient, such as a dropped message, retrying the operation is often sufficient. If, however, the failure is due to inconsistent state among the replicas, then no progress can be made until a new configuration is created by the `repair()` protocol. As described in Section 7.3, the client library continues sending the request, reading the latest state of the system via `get_cert()`, and performs repair audits until the request is either sound or the state of the system has changed indicating the

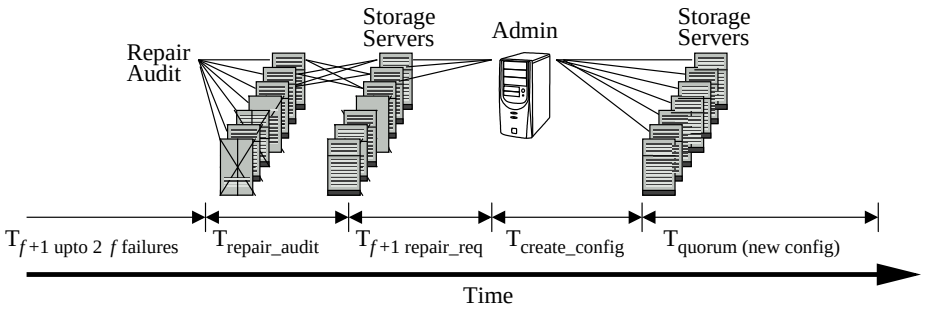


Figure 8.8: When a storage server believes that repair is needed, it sends a request to the administrator. After the administrator receives $2f+1$ requests from servers in the current configuration, it creates a new configuration and sends message to servers in the set. The message describes the current state of the log; storage servers fetch the log from members of the previous configuration.

request is unsound.

8.5.4 The `repair()` Protocol

The `repair()` protocol is used to restore the replicated log to a consistent state on a sufficient number of servers. It may be used when a client cannot make progress because replicas of the log are in an inconsistent state or a quorum is not available due to server failures. Figure 8.8 shows the repair process.

A repair audit (Section 8.4.5) causes a storage server to check the replicated state and availability of servers in a configuration. Repair audits can be triggered by any component, client, storage server, administrator, periodic timer, etc. However, only storage servers in the latest configuration can trigger the actual repair protocol. Repair requests from storage servers not in the latest configuration are ignored by the administrator.

Repair begins when the administrator receives signed repair requests from at least $2f+1$ storage servers in the latest configuration. A repair request is a signature over the soundness proof and current configuration. The signature is only valid if it is from a storage server in the latest configuration.

When the administrator receives $2f+1$ repair requests, it creates a new configuration to host the log. The administrator determines the current state of the log by extracting the latest soundness proof from the $2f+1$ submitted requests. This is guaranteed to reflect the latest sound write (Theorem 1). The administrator sends the new configuration and latest soundness proof to the storage servers in the new configuration. Servers in the new configuration fetch the log (defined

in the soundness proof) from servers in the previous configuration. The administrator can reduce the amount of data that must be transferred during repair by retaining servers across configurations. After acquiring a copy of the log, a storage server in the new configuration responds to the administrator with a signature over the certificate (contained in the latest proof) and the new configuration. After the administrator receives a quorum of responses from servers in the new configuration, it can create a soundness proof in the new configuration.

Next, the administrator sends the soundness proof to all the servers and waits for acknowledgment. When the servers receive the soundness proof, they record it to stable storage and respond to the administrator. After receiving acknowledgment from a quorum of servers, the administrator can be certain the log has been repaired.

The administrator continues to resend the message notifying a storage server that it has been assigned to a configuration until one of two conditions is met. When the administrator receives a reply from a quorum of servers in the new configuration, it can be certain that the configuration can accept new requests. Alternatively, if the administrator receives $2f+1$ requests from servers in the new configuration to repair the configuration, it restarts the repair protocol. Because a new configuration has at most $2f$ faulty servers, the administrator is assured that one of these conditions will be met eventually.

We explain how we relieve the administrator of notifying storage servers of their new role in Part IV. Briefly, the administrator selects a server to coordinate repair. The coordinator contacts all the servers in the new configuration, creates a soundness proof, and informs the administrator when repair is complete. As a result, the administrator is responsible for creating a new configuration and the coordinator executes the remaining repair protocol.

8.5.5 Transient Server Failure Protocol

A server executes a transient failure protocol to update its local state to the latest sound write. The protocol may be executed by a server at any time. For instance, a server may execute the protocol periodically, after receiving evidence that its local state is out-of-date (e.g. after receiving a more recent soundness proof in a request), or after returning from failure as part of the recovery protocol.

The transient failure protocol has two steps. First, a server queries $2f+1$ servers using the `get_cert()` interface with the option that the response includes the latest soundness proof stored by the server and block_names. $2f+1$ replies are sufficient to discover the latest sound write assuming

at most $2f$ servers are faulty and at most f of those faulty servers are malevolent. This first step can be skipped if the server received a later soundness proof in a request. Second, if the server's state is in fact out-of-date, it selects one of the servers that responded with the latest proof and uses the `get_blocks()` interface to fetch the missing blocks from that server. If the `get_blocks()` fails, the server can select another server that responded to the `get_cert()` or one that signed the soundness proof to fetch the missing blocks. Finally, the server's state is up-to-date after executing both steps.

8.6 Protocol Correctness

We now demonstrate that the protocol presented in Section 8.5 satisfies the requirements in Section 8.2. First, we show that sound writes define a total order satisfying *consistency*. Second, we show that $2f+1$ servers that exist in a configuration are sufficient to trigger repair and ensure *durability*. Finally, we show that write attempts can eventually succeed providing *liveness*.

8.6.1 Protocol Consistency

Theorem 2. *Given a configuration size of $n = 4f+1$, quorum size of $q = 3f+1$, and $2f$ faulty servers with at most f malevolent, each sound write has a unique certificate and configuration sequence number pair.*

Proof. Proof by contradiction. Assume two clients submit a write attempt with the same sequence number, but conflicting data (data and resulting verifier differ). Assume further that both clients receive sufficient replies for a sound write implying a quorum $q = 3f+1$ positively acknowledged both client's requests. This is not possible since the intersection between both write attempts is at least $2f+1$ servers which is greater than the number of malevolent servers f. □

Theorem 2 proves that each sound write has a unique position in the set of sound writes. The unique position defines a total order of sound writes, which is sufficient to maintain consistency [LSP82, Sch90].

8.6.2 Protocol Durability

Theorem 3. *Given a configuration size of $n = 4f+1$, quorum size $3f+1$, and $2f$ faulty servers with at most f malevolent, sound writes exist as long as $2f+1$ servers in a configuration exist.*

Proof. $2f+1$ servers are sufficient to trigger repair protocol (Theorem 1), which copies data up to the latest sound write to a new configuration. □

Theorem 3 proves that sound writes persist as long as they can be repaired.

8.6.3 Protocol Liveness

As presented, the protocol ensures consistency and durability, but not liveness in the face of continually conflicting write attempts. For example, it is possible that the replicated state continually needs repair starving write attempts from succeeding. However, the protocol does ensure that write attempts eventually succeed. In the absence of conflicting writes and repair, we show from any "wedged" state, the replicated state can be repaired and subsequent write attempts can eventually succeed.

Theorem 4. *Given a configuration size of $n = 4f+1$, quorum size $3f+1$, and $2f$ faulty servers with at most f malevolent, write attempts eventually succeed.*

Proof. Invoke repair. Repair installs a new configuration with the latest sound write (Theorem 1). In the absence of conflicting writes, re-applying the write attempt against the new configuration will succeed. □

8.7 Discussion

In this section, we discuss how a secure log reduces the complexity of implementing a dynamic Byzantine quorum.

First, self-verifying data in a secure log reduces server attacks. In particular, there are two server attacks, not responding to queries and responding with old values. The protocol tolerates the first attack by allowing writes to succeed without responses from up to f servers. Additionally, the protocol ensures consistency and durability tolerating up to $2f$ faulty servers that may respond with old values.

Second, most of a secure log is stored as immutable (read-only) hash-verified extents reducing the number of operations that modify state. `snapshot()` or `put()` can create a hash-verified extent and `repair()` is the only allowed operation after it has been created. `repair()` changes the configuration but cannot change the contents of the extent.

Third, the secure log structure reduces the complexity of updating the replicated log. In particular, the new local state stored by an up-to-date server is always derivable from applying the request to the current local server state. For example, for an `append()`, the following field and function are equivalent

```
pending_proof.certificate.verifier =
    computeVerifier(pending_block_names[ ], proof.cert.verifier)
```

See Figure 8.6 for a description of a server's local state and Figure 6.2 for a description of computing the verifier.

Fourth, the secure log structure reduces the complexity of maintaining sound writes over-time. Only the latest soundness proof and associated data is maintained since the state of the latest sound write can be derived from previous sound writes and data.

Finally, a secure log reduces the complexity of the transient failure protocol. A server needs to only query the other servers to find the latest sound write, then fetch the missing blocks from one of the up-to-date servers.

In summary, the secure log structure and narrow interface simplify designing a consistency protocol such as a dynamic Byzantine quorum protocol. More importantly, the secure log reduces the complexity of implementing the replicated state protocols that maintain the distributed secure log. Each write or repair operation modifies the secure log in a well-defined manner where invariants of the local and replicated state can be checked at each step of the protocol. Part IV demonstrates that a secure log reduces the complexity of implementing, replicating, distributing replicas, and maintaining consistency over replicas.

Chapter 9

Utilizing Distributed Hash Table (DHT) Technology for Data Maintenance

A distributed wide-area on-line archival storage system requires a self-organizing mechanism to locate extents and trigger repair audits as servers fail. In this section, we describe the architecture of an extent replica location and repair service (Part II) implemented as a distributed directory. A distributed directory allows each server to be part of the directory and collectively provides a data maintenance service. At its core, a distributed directory is a level of indirection– utilizing pointers within the network to achieve flexibility in data placement, locating extents, and timely repair based on low watermarks. We describe all of the components necessary to implement a distributed directory: publishing and locating extent replicas, monitoring server availability and triggering repair audits.

This distributed directory architecture has a large scope where all servers are eligible to store replicas. As demonstrated in Section 4.3.1, a large scope reduces repair time since more servers can assist in repair. The decrease in repair time increases durability since durability is inversely proportional to repair time [PGK88]. The rest of this chapter describes the distributed directory architecture and is organized as follows. In Section 9.1, we describe how servers use the distributed directory to publish and lookup the location of extents. We describe how the distributed directory monitors server availability and triggers a repair audit in Section 9.2. Finally, in Section 9.3, we discuss the limitations of a distributed directory.

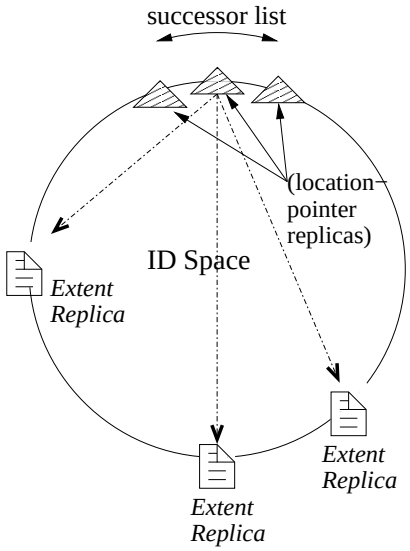


Figure 9.1: Distributed Directory System Architecture.

9.1 Publishing and Locating Extent Replicas

Publishing and locating extent replicas is a prerequisite for an on-line archival storage system. It works by using a structured overlay that supports Key-Based Routing [DZD+03] (KBR) to route messages; such overlays include Bamboo [RGRK04], Chord [SMK+01], Pastry [RD01], and Tapestry [ZHS+04]. KBR works by consistently hashing an identifier space over a set of servers. Each of these servers is responsible for a continuous interval of the identifier space known as the *root*. An identifier is assigned to each server using a secure hash function like SHA-1. For example, in Bamboo and Pastry, the server whose identifier is numerically closest to the object's identifier (in the identifier ring), is responsible for the object. A server participating in the structured overlay usually maintains $O(\log N)$ state (i.e. routing table) and the path from a server to the root takes $O(\log N)$ hops. Structured overlays often implement a distributed hash table (DHT) interface, `put()` and `get()`, where the root server stores an object on a `put()` request and returns an object on a `get()` request. We use the DHT interface to store and retrieve location-pointers.

`Publish()` is analogous to a `put()` operation into a DHT where the value stored by the DHT is a location-pointer that maps the replica identifier to the storage server storing the replica. `Lookup()` is another operation. `Lookup()` is analogous to a `get()` operation from a DHT where the values returned by the DHT are location-pointers.

Implementing `Publish()` and `Lookup()` using a structured overlay was first proposed

```
SELECT location-pointers
FROM  p AS pointerDB
WHERE p.objId=objId AND
      p.low_watermark >
       (SELECT COUNT
        FROM p2 as pointerDB,
             s AS serverAvailDB
        WHERE p2.objId=p.objId  AND
              p2.src=s.serverId AND
               s.state=UP)
```

Figure 9.2: The above query states that for a given object identifier select the location-pointers where the remaining number of replicas are less than the low_watermark, thus triggering a repair audit

by Tapestry [HKRZ02, ZJK01]. The abstraction is called Distributed Object Location and Routing (DOLR). A distributed directory is different to a DOLR in two ways. First, a distributed directory is optimized for maintenance and not routing. Second, we have extended the DOLR to efficiently maintain large quantities of location-pointers over long periods of time.

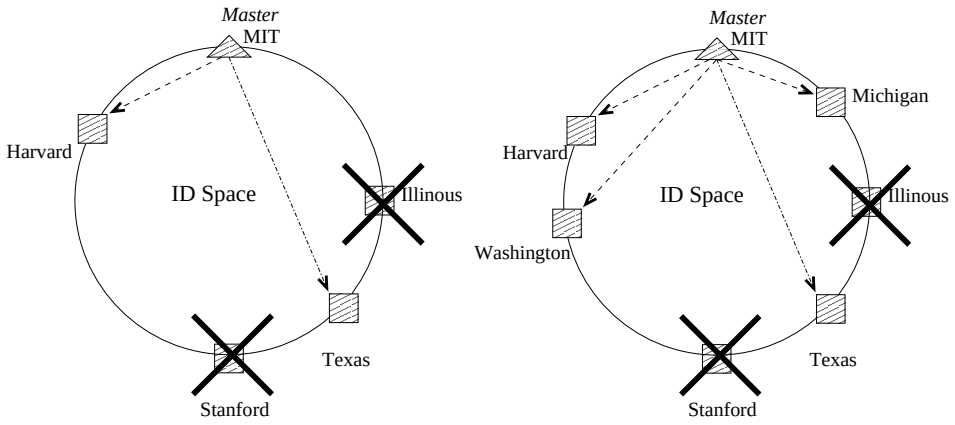
9.2 Monitoring Server Availability and Triggering Repair Audits

To maintain extents, each storage server also serves as a monitoring server. Long term data maintenance requires that each monitoring server know the number of available extent replicas of each extent for which it is responsible. The goal of monitoring is to allow the monitoring servers to track the number of available extent replicas and to learn of extents that the server should be tracking but is not aware of. When a monitoring server n fails the new server n' that assumes responsibility of n's location-pointers begins tracking extent replica availability; monitored information is soft state and thus can be failed over to a "successor" relatively transparently.

In a distributed directory, replicas for a particular extent are stored on different storage servers and the DHT stores and maintains replica location-pointers. Namely, each extent replica (for a particular extent) has the same object identifier[1], but a different location. As a result, a particular extent has a unique monitoring server called a *root* that resolves replica location requests and triggers a repair audit to replace replicas lost to failure. Figure 9.1 shows a directory system architecture.

The root monitoring server stores the status of all storage servers in a server availability

[1]Each fragment for a particular extent has the same identifier if we use erasure codes [WWK02].



(a) Threshold Reached (b) Data Recovered

Figure 9.3: Directory Data Recovery. a) Using its location-pointers and storage server availability database, the root monitoring server (MIT) knows that there are two replicas remaining. If the low watermark is three, then the root triggers a repair audit. b) The storage servers containing the remaining replicas (Harvard and Texas) cooperate to refresh lost data replicas.

database. It updates the server availability database by receiving heartbeats from all storage servers; this message includes a generation identifier that is randomly generated when the server is installed or reinstalled following a disk failure. It interprets the lack of receiving a heartbeat as a storage server failure signal. After receiving a failure signal, it triggers a repair audit for every extent stored on the newly down server where the number of available replicas is below the low watermark. When a server returns from failure, sends a heartbeat, and its generation identifier has not changed, the monitoring server can conclude that objects stored on that server are again accessible increasing the available replicas for extents stored on the server.

To determine if a repair audit needs to be triggered, a monitoring server performs a join between the location-pointer and server availability databases and selects all extents where the number of available servers is less than the low watermark. Figure 9.2 is an example of the database code that a root monitoring server uses to determine it needs to trigger a repair audit. Every failure can potentially cause a scan of the location-pointer and server availability database; as a result, scans are run periodically to reduce load on the databases.

When the available replication level is below the low watermark, the root monitoring server informs the remaining storage servers that a replica has been lost via a repair audit. After receiving a repair audit, the remaining storage servers cooperate to refresh the lost replica. Figure 9.3.(a) shows an example of a root monitoring server with two remaining available replica

location-pointers. If the low watermark is three, then the root monitoring server triggers a repair audit since the number of available replicas is less than the low watermark. The root monitoring server sends a repair audit to the remaining replica storage servers indicating that the number of replicas is below the low watermark. Figure 9.3.(b) shows the remaining storage servers cooperating to refresh the lost replicas.

9.3 Discussion and Limitations

Designing a self-organizing and maintaining monitoring system is required for distributed wide-area on-line archival storage. However, its design and associated costs need to be carefully considered. We discuss implied assumptions and consequences below.

First, when using a distributed directory, we assume the storage due to location-pointers is significantly less than storage due to data. The implication of this first assumption is that replicas are large (e.g. $1MB$) when using a directory; otherwise, if replicas were small (e.g. $8kB$) and the location-pointers were further replicated, then the storage system would maintain more data location state than data itself. Furthermore, we assume the total number of replicas of all extents is greater than the number of servers. The implication of this second assumption is that it is more feasible to maintain a storage server availability database on every monitoring server than it is to republish the location of every replica. For example, it is cheaper to perform an all-pairs ping than to republish[2] every replica for a storage system with 1000 servers, 1TB of storage per server, 1MB extents, and a replication factor of 10 (i.e. 10 billion total replicas or 10 million replicas per server vs. 1000 servers). If either assumption is violated then a distributed directory should not be used.

Second, monitoring servers limits scalability. A distributed directory has to monitor all N storage servers. It can still be efficient because replicas per server is assumed to be much greater than the number of servers; however, care must be taken when designing a monitoring system. For example, if each directory root monitoring server is also a storage server, than sending and receiving heartbeats is the same as performing an all-pairs ping. The cost of an all-pairs ping per server is dependent on the number of servers N, the heartbeat period T_{hb}, and the size of a heartbeat hb_{sz} (i.e. $\frac{BW_{hb}}{N} = \frac{N}{T_{hb}} \cdot hb_{sz}$). The resulting N^2 probe traffic may limit the system's scalability.

An alternative implementation to update the server availability database is an all-pairs multicast. Each storage server sends a heartbeat to its neighbors with expanding radius; that is, servers at a further radius receive heartbeats less often than servers at a closer radius. The DHT's

[2] A republish updates the location-pointers for a directory and prevents them from expiring.

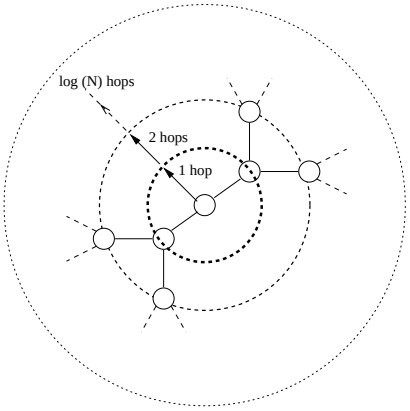


Figure 9.4: Expanding Radius Heartbeat. Heartbeats initiated by a storage server (e.g. middle server) reach a greater number of additional servers as the heartbeat radius expands. Heartbeats are a form of multicast and reach all servers in the system when the radius is $\log N$.

routing tables are used to establish a unique spanning tree rooted at each server with $O(\log N)$ out-degree per server. Each server periodically broadcasts a heartbeat message to its children in the tree. The children rebroadcast the heartbeat to their children until it is received by all servers. Heartbeats are received by a neighbor at radius-i with a period $p_i = p_{i-1} * f$, where p_1 (the base period) and f (the factor by which the periods grow exponentially) are configurable parameters. Note that when f is 1, all neighbors in the network receive heartbeats at the same rate, regardless of their radius. Figure 9.4 shows an example of an expanding radius heartbeat. We use an expanding radius heartbeat in Antiquity in Part IV.

Part IV

Antiquity: Prototype Implementation and Evaluation

Chapter 10

Antiquity

Thus far in this thesis we have explored the design space for fault tolerant and repair algorithms designed to efficiently maintain data durability. Furthermore, we have described an architecture that applies these design principles to a log based on-line archival storage system. The design is secure and expressive enough to support applications. In this part of the thesis we further answer the question: is it possible to build a self-organizing and maintaining distributed wide-area storage system that supports a secure append-only log? How does such a system perform?

To verify the design, we have built and deployed a secure distributed wide-area storage system based on a log called Antiquity. Antiquity integrates the above design points into one cohesive system. Antiquity, uses a distributed hash table (DHT) as an underlying layer to connect storage servers. It optimally maintains immutable data by reintegrating extant replicas. Further, it efficiently maintains order integrity of mutable data of the log head by implementing dynamic Byzantine quorum and quorum repair algorithms for consistency and durability. Finally, it implements the secure append-only log interface for usability.

Experience with the prototype shows that Antiquity's design is robust. It has been running for over two months in the wide-area on 400+ PlanetLab [BBC+04] servers maintaining nearly 20,000 logs containing more than 84 GB. Periodic random reads reveal that 94% of all replicated logs are in a consistent state. 100% of the logs are durable, though 6% do not have a quorum of servers available due to transient failures. This matches the expected number of consistent quorums, assuming an average PlanetLab server availability of 90% and given that a quorum requires the availability of five out of seven servers. The prototype maintains a high degree of consistency and availability due to the quorum repair protocol despite the constant churn of servers (a quarter of the servers experience a failure every hour).

10.1 Architecture Overview

The Antiquity prototype implementation combines the dynamic Byzantine quorum, quorum repair, and aggregation discussed in Part III with fault tolerance and repair techniques discussed in Part II. Chapter 9 discussed how the system utilizes self-organizing distributed hash table (DHT) technology.

In review, the storage system supports a secure, append-only, log abstraction where a single log is owned by a single principal identified by a cryptographic key pair. Only the owner can `append()` to the log. The storage system stores the log as a sequence of container objects called extents, where each extent stores a variable number of arbitrarily-sized application-level log elements. To guard data integrity, individual log elements and whole extents are self-verifying.

A log is composed of two types of extents. The log head is a mutable, key-verified extent; all other extents are immutable hash-verified extents. The key-verified log head is named by a secure hash of the public key associated with the log. To verify the contents of the log head, a server compares the data to the verifier included in an associated certificate (after confirming the signature on the certificate). A hash-verified extent is named by a function of the contents of the extent. A server can verify the integrity of a hash-verified extent by comparing an extent's contents to its name.

An extent is the unit of storage and replication. The storage system maintains consistency of key-verified extents and durability of both key- and hash-verified extents. Each server participating in the system serves as a storage server for extents. The storage server is implemented as a state machine for each extent that it stores. To modify state, the storage servers implement the API enumerated in Table 7.2. The system determines the true state of an extent using the quorum protocols of Chapter 8. Finally, the Antiquity prototype efficiently maintains durability using fault tolerant and repair algorithms discussed in Part II.

The prototype is implemented in 41,000 lines of Java code and uses BerkeleyDB to store state to local disk. The client library is an additional 3,000 lines of code. All servers in the system communicate via RPC using a custom library written in Java that takes advantage of Java's framework for asynchronous I/O. For cryptographic operations, the implementation uses GNU's multiprecision library with 1024-bit RSA keys.

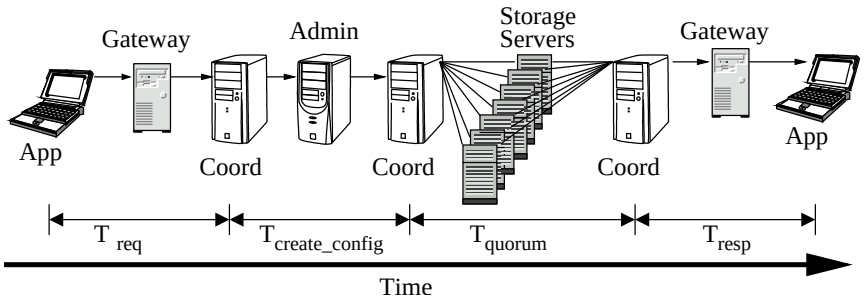
The Bamboo DHT [RGRK04] underlies and connects the storage servers. We use the DHT as a distributed directory; that is, the DHT does not store data, but rather it stores pointers that identify servers that store the data. A distributed directory provides a level of indirection that

allows flexible data placement which can increase the durability and decrease the cost of repairing a given replica [CDH+06, vS04]. The storage servers use the distributed directory to publish and locate extent replica locations. Additionally, the storage servers also use the DHT in the traditional manner as a storage cache to reduce load on storage servers and the administrator. In particular, Antiquity uses the DHT as a cache for soundness proofs (proof that a write is consistent and durable, see Section 8.4.4) to ensure they are available for all interested parties.

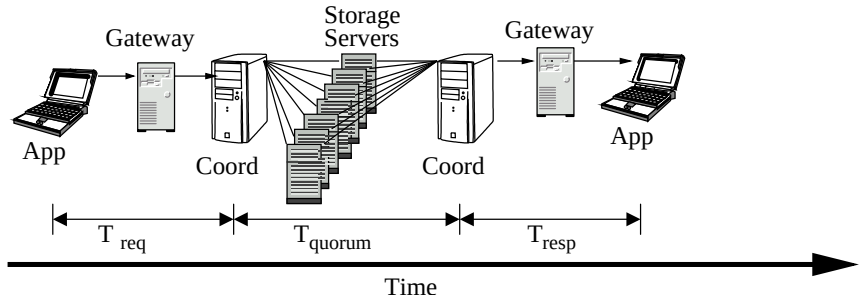
The prototype also relies on the DHT to help monitor server availability to determine when repair is necessary. Using the DHT as a distributed directory, it is not efficient to monitor the availability of each extent separately. Instead, we use the DHT to monitor server availability and use that metric as a proxy for extent availability. To monitor server availability, we use a scheme that periodically broadcasts a heartbeat message through a spanning tree defined by the DHT's routing tables [CDH+06]. A monitoring server receives liveness information from each storage server with a frequency depending on its distance in the spanning tree. Additionally, it sends a repair audit if it fails to receive an expected heartbeat.

In our implementation, each server in the system also serves as a *gateway*. A gateway accepts requests from a data source and works on behalf of that source. It determines the configuration, set of storage servers responsible to handle the request. Further, it multicasts the request to the configuration and aggregates responses. The use of the gateway lowers the bandwidth requirements of the data source. Because all requests are signed and all data is self-verifying, inserting the gateway in the path between the data source and the storage servers does not affect security. If the data source believes a failure is due to a faulty gateway, it can resend the request through a different gateway. To make the soundness proof available to storage servers earlier, the gateway combines responses from the storage servers to create a proof and publishes that proof in the DHT. A soundness proof is proof that a write is consistent and durable.

The administrator, required by the design, is implemented as a single server. We take several measures to limit the work that must be performed by the administrator. First, the administrator does not *create* new configurations. Instead, it *verifies* configurations proposed by the gateways. The administrator requires a configuration be created based on the neighbor lists of the underlying DHT, which is effectively random. It compares the proposed configuration against neighbor lists from the underlying DHT before signing, and thus authorizing, the configuration. To limit the number of configuration queries that an administrator must handle, other machines in the system can cache valid configurations. Finally, to avoid the burden of notifying servers of their membership in a new configuration, the administrator sends notice of new configurations to the gateway that is han-



(a) `create()`/`put()`/`snapshot()`/`renew()` path



(b) `append()`/`truncate()` path

Figure 10.1: The path of an (a) `create()`/`put()`/`snapshot()`/`renew()` and (b) `append()`/`truncate()`.

dling the request that requires the new configuration. The gateway is responsible for informing and multicasting the message to the new configuration. We run the administrator server on a machine at our site in Berkeley.

We discuss these components in more detail in the following sections.

10.2 Gateways, Coordinators, Distributed Hash Tables, and Protocol Details

To understand the performance of Antiquity, we must understand the complete path for each request. First, a data source sends a request to a gateway via remote procedure call (RPC). Second, the gateway multicasts the request to the storage servers and aggregates responses (possibly contacting the administrator first for operations such as `create()`). Finally, the gateway forwards the responses to the source as proof that the request is consistent and durable. Figure 10.1 illustrates a typical path for `create()` and `append()`.

If many data sources use the same gateway, however, the gateway can become over-

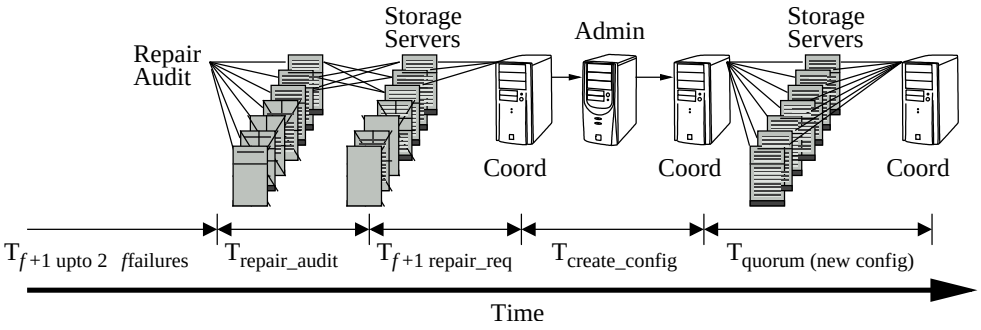


Figure 10.2: The path of an (a) `repair()`

whelmed. Thus, for load balance in implementation, the gateway forwards the request to a *coordinator*. The coordinator performs the multicast and aggregation. The coordinator is the root of the identifier space. This implementation design decision was a response to observed overload of gateways in implementation and has some negative consequences and benefits. The obvious consequence is that a coordinator adds latency to a request. There are three benefits, however. First, the coordinator effectively reduces the load of a gateway. The number of extents each coordinator is responsible for is proportional to the number of servers. Each coordinator, identified as the root of the identifier space, must multicast requests and aggregate responses for particular extents it is responsible. Second, some of the added latency can be reduced since the coordinator is often a storage server for the request since configurations are chosen based on the neighbor space. Finally, recall that inserting the gateway and/or coordinator in the path between the data source and storage servers does not affect security since storage servers do not possess the source's private key.

10.2.1 Path of a `create()`, `append()`, and `repair()`

We now review the path of a `create()` and `append()` as canonical examples that do and do not involve the administrator. Figures 10.1.(a) and (b) illustrates these paths. Additionally, we review the path of a `repair()` illustrated in Figure 10.2.

First, a `create()` proceeds from the data source to a coordinator through a gateway. This process involves a DHT `lookup()` and `publish()` to locate and publish the coordinator. The coordinator is the root of the identifier space. Next, the coordinator forwards the request to the administrator to create a configuration. After the administrator creates a configuration and responds, the coordinator multicasts the `create()` and configuration to the storage servers. The storage servers sign that they received and applied the request. Next, the coordinator creates a soundness proof

T_{req}

`create()`	$\mathbf{s}_{cl} + \mathcal{N}(gw\_req)_{cl\_gw} + [v + \text{L}]_{gw} + \mathcal{N}(gw\_req)_{gw\_co} + [v + 2 \cdot \text{L} + \text{P}]_{co}$
`append()`	$\mathbf{s}_{cl} + \mathcal{N}(gw\_req)_{cl\_gw} + [v + \text{L}]_{gw} + \mathcal{N}(gw\_req)_{gw\_co} + v_{co}$

$T_{\text{create\_config}}$

`create()`	$\text{L} + \mathcal{N}(ad\_req)_{co\_ad} + [v + \mathbf{s} + 2 \cdot \mathcal{D}(ad\_req)]_{ad} + \mathcal{N}(ad\_resp)_{ad\_co}$

T_{quorum}

`create()`	$n \cdot \mathcal{N}(ss\_req)_{co\_ss} + [2 \cdot v + 2 \cdot \mathcal{D}(ss\_req) + \mathbf{s} + \text{P}]_{ss} + \mathcal{N}(ss\_resp)_{ss\_co} + [n \cdot v + \text{P}]_{co}$
`append()`	$n \cdot \mathcal{N}(ss\_req)_{co\_ss} + [v + 2 \cdot \mathcal{D}(ss\_req) + \mathbf{s}]_{ss} + \mathcal{N}(ss\_resp)_{ss\_co} + [n \cdot v]_{co}$
`snapshot()`	$n \cdot \mathcal{N}(ss\_req)_{co\_\text{newss}} + [2 \cdot v + \mathcal{D}(ss\_req)]_{\text{newss}} +$ $\mathcal{N}(\text{olds}s\_req)_{\text{newss\_olds}s} + \mathcal{D}(\text{olds}s\_req)_{\text{olds}s} + \mathcal{N}(\text{olds}s\_resp)_{\text{olds}s\_\text{newss}} +$ $[\mathcal{D}(\text{olds}s\_resp) + \mathbf{s} + \text{P}]_{\text{newss}} + \mathcal{N}(ss\_resp)_{\text{newss\_co}} + [n \cdot v + \text{P}]_{co}$

T_{resp}

`create()`	$\mathcal{N}(gw\_resp)_{co\_gw} + \mathcal{N}(gw\_resp)_{gw\_cl} + [n \cdot v]_{cl}$
`append()`	$\mathcal{N}(gw\_resp)_{co\_gw} + \mathcal{N}(gw\_resp)_{gw\_cl} + [n \cdot v]_{cl}$

Table 10.1: Breakdown of latencies for all operations. Unless an operation is stated explicitly, `create()` represents all operations that interact with the administrator such as `put()`/`snapshot()`/`renew()`, and `append()` represents all operations that do *not* such as `truncate()`. Total operation latency is $T_{\text{req}} + T_{\text{create\_config}} + T_{\text{quorum}} + T_{\text{resp}}$ for `create()` and $T_{\text{req}} + T_{\text{quorum}} + T_{\text{resp}}$ for `append()`. For all time breakdowns $\mathcal{N}(X)_{a\_b} = (\alpha_{net} + X\beta_{net})$ and $\mathcal{D}(X) = (\alpha_{disk} + X\beta_{disk})$ are the network (from a to b) and disk delays, respectively, where α is the latency, β is the inverse of the bandwidth (bytes per second), and X is the number of bytes. Next, cl =client (or app), gw = gateway, co = coordinator, ad = administrator, and ss = storage server. Finally, $\mathbf{s}$, v, L, and P are the times to sign, verify, DHT `lookup()`, and DHT `publish()`, respectively. Notice that `create()` requires three signatures and `append()` requires two.

from a quorum of the storage servers responses. The soundness proof certifies that the `create()` is consistent and durable. Then, the coordinator publishes the soundness proof in the DHT. Finally, the coordinator sends the soundness proof included in a response to the data source via the gateway. Figure 10.1.(a) shows the path of `create()`.

Second, the path of an `append()` is similar to `create()` except the administrator is skipped. Additionally, the DHT `lookup()` and `publish()` can be skipped if the extent has recently been read or written. As a result, an `append()` is directly sent from the data source to the coordinator via the gateway. The coordinator multicasts the `append()` to the storage servers. The `append()` includes a soundness proof from the previous `create()` operation. In general, `append()` includes a soundness proof of the previous operation, which serves as a predicate for the new `append()`. When the storage servers receive an `append()` request, they verify the request, and sign that they received

and applied the request. Next, the coordinator creates a new soundness proof from a quorum of storage server responses. The soundness proof certifies that the `append()` is consistent and durable. Finally, the coordinator simultaneously publishes the soundness proof into the DHT and sends it to the data source via the gateway. Figure 10.1.(b) shows the path of `append()`

Finally, the path of a repair audit and `repair()` includes a coordinator as well. The coordinator, however, is often one of the storage servers since the coordinator is the root of the identifier space and the storage servers in the configuration are chosen based on neighbors in the identifier space. Upon receiving a repair audit, the coordinator collects signed repair statements from the storage servers in response to a repair audit. If $2f+1$ or more signed repair statements are received, the storage server forwards the repair statements to the administrator. The administrator creates a new configuration and sends it to the coordinator to multicast to the new storage servers. The coordinator can be skipped and the storage servers can communicate with the administrator directly. We use the coordinator for increased efficiency to remove load from the administrator. Again, the coordinator poses no security risk since it cannot sign configurations and the administrator can pick a new coordinator if one is not being responsive. Figure 10.2 shows the path of a repair audit and `repair()`.

10.2.2 Breakdown of latencies for all operations

In this section we discuss the notation and breakdown of latency for operations. Table 10.1 shows the breakdown for operations that do and do not involve the administrator. Unless an operation is stated explicitly, `create()` represents all operations that interact with the administrator such as `put()`, `snapshot()`, and `renew()`; `append()` represents all operations that do *not* such as `truncate()`.

The latency notation is as follows. Total latency for an operation that interacts with the administrator such as `create()` is the sum of $T_{\text{req}} + T_{\text{create\_config}} + T_{\text{quorum}} + T_{\text{resp}}$, which represents the time for a client to send a request, system to create a configuration, send and receive acknowledgments from a quorum, and forward the result back to the client. An operation that does not interact with the administrator such as `append()` is similar to a `create()` except it does not include the $T_{\text{creae\_config}}$ term; as a result, the total latency is $T_{\text{req}} + T_{\text{quorum}} + T_{\text{resp}}$ for `append()`.

Each request includes a number of interactions with the network and disk. The network and disk delays are represented with the following notation. $\mathcal{N}(X)_{a\_b} = (\alpha_{net} + X\beta_{net})$ is the network delay from a server a to a server b. α is the latency, β is the inverse of the bandwidth (bytes

per second), and X is the number of bytes. Similarly, for disk, $\mathcal{D}(X) = (\alpha_{disk} + X\beta_{disk})$ is the disk delay (α, β, and X have the same meaning as they do for the network notation).

The final set of notations represents the communicating components and time to perform some auxiliary functionality. The communicating components include the cl = client (or app), gw = gateway, co = coordinator, ad = administrator, and ss = storage server. auxiliary routines include the times to sign, verify, DHT `lookup()`, and DHT `publish()` which are represented by $\mathbf{s}$, v, L, and P, respectively.

The T_{req} term in Table 10.1 includes the time for a client to sign a request and send it to the gateway of the storage system. The gateway verifies the request and performs a DHT `lookup()` to locate the storage servers and coordinator. If a coordinator already exists, then the gateway forwards the request; otherwise, it performs a DHT `publish()` to announce that it has become the coordinator followed by a DHT `lookup()`. The `append()` operation often does not need to perform any DHT operations since servers cache the location of the coordinator and storage servers.

The $T_{\text{create\_config}}$ term in Table 10.1 includes the time for the coordinator to identify the list of neighbor servers to use in a configuration via a DHT `lookup()`. Furthermore, it includes the time for the coordinator to forward the configuration creation request to the administrator. Next, it includes the time for the administrator to verify the request, validate the neighbor list for the new configuration, perform a disk read and write, and sign the new configuration. Finally, it includes the time for the administrator to respond with the new configuration. Note that only `create()` operations interact with the administrator.

The T_{quorum} in Table 10.1 includes the time for the coordinator to multicast the operation to all the storage servers in the configuration. Additionally, it includes the time for the servers to verify the request, perform a disk read and write, sign its response, and `publish()` its location (storage servers invoke the DHT `publish()` only for the `create()`, `snapshot()`, and `put()` operations). Next, it includes the time for the coordinator to receive and verify the responses from a quorum of servers and `publish()` the resulting soundness proof. Notice, that `snapshot()` additionally includes the time for the new snapshot servers to fetch the head of the log. This fetch includes the time for the new snapshot server to contact a remote server storing the head of the log, time for the remote server to read the data from disk, and time for the remote server to respond.

Finally, the T_{resp} term in Table 10.1 includes the time for the coordinator to respond to the client through the gateway. Furthermore, it includes the time for the client to verify the soundness proof contained in the response.

10.3 Summary and Discussion

The performance observed throughout Chapter 11 reflects the intrinsic properties of the design.

One feature inherent in the design is the number of signatures required per operation. The use of quorums keeps the number of required signatures low. The data source must sign a certificate for each request that modifies the state of the log. Each storage server that handles such a request must sign its response. Because the `create()`, `snapshot()`, and `put()` requests require a new configuration, these operations also require the administrator to create one signature. Note, the most common operation, `append()`, does not require the additional administrator signature. The number of signatures required in Antiquity matches the number required of other systems that tolerate Byzantine failure [REG+03]. In particular, operations that involve the administrator require three signatures and those that do not involve the administrator require two.

Another integral aspect of the design is the number of messages transmitted to handle a request. The quorum-based Antiquity system uses $O(n)$ messages between coordinator and storage servers; the storage servers do not exchange any messages. To obtain a new configuration for `create()`, `snapshot()`, and `put()` operations, the gateway must send an additional message to the administrator. The number of messages required in Antiquity compares favorably to other designs based on Byzantine agreement protocols that require $O(n^2)$ between servers.

Byzantine quorums also limit the number of *rounds* of messages to complete a request. Quorums require one round of communication; the coordinator sends a message to each storage server, and each server responds. Byzantine agreement algorithms, on the other hand, require three rounds of messages between storage servers. By limiting the number of rounds, the quorum-based design reduces the number of network round trips.

Finally, the design maintains consistency and durability of the log even though a quorum may not immediately be available. In particular, a quorum repair algorithm restores the log to a consistent and available state. The consistency and durability come from the soundness proof which ensures that a sound write is durable as long as $2f + 1$ storage servers exist(Theorem 1).

Chapter 11

Evaluation

In this chapter, we present results from testing the Antiquity prototype deployed on PlanetLab and a local cluster. We focus our evaluation on the primitive operations provided by the storage system, but we also describe our experiences with a versioning archival back-up application.

11.1 Experimental Environment

We are currently running two separate Antiquity deployments. Both deployments are configured to replicate each extent on a configuration of seven storage servers (except where explicitly stated otherwise). Thus, each configuration can tolerate *two* faulty servers in each configuration. Both deployments are hosted on machines shared with other researchers, and, consequently, performance can vary widely over time.

The first deployment runs on 60 nodes of a local cluster. Each machine in the *storage cluster* has two 3.0 GHz Pentium 4 Xeon CPUs with 3.0 GB of memory, and two 147 GB disks. Nodes are connected via a gigabit Ethernet switch. Signature creation and verification routines take an average of 3.2 and 0.6 ms, respectively. This cluster is a shared site resource; a load average of 5 on each machine is common.

The other deployment runs on the *PlanetLab* distributed research test-bed [BBC+04]. We use 400+ heterogeneous machines spread across most continents in the network. While the hardware configuration of the PlanetLab nodes varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. Signature creation and verification take an average of 8.7 and 1.0 ms, respectively. PlanetLab is a heavily-contended resource and the average elapsed

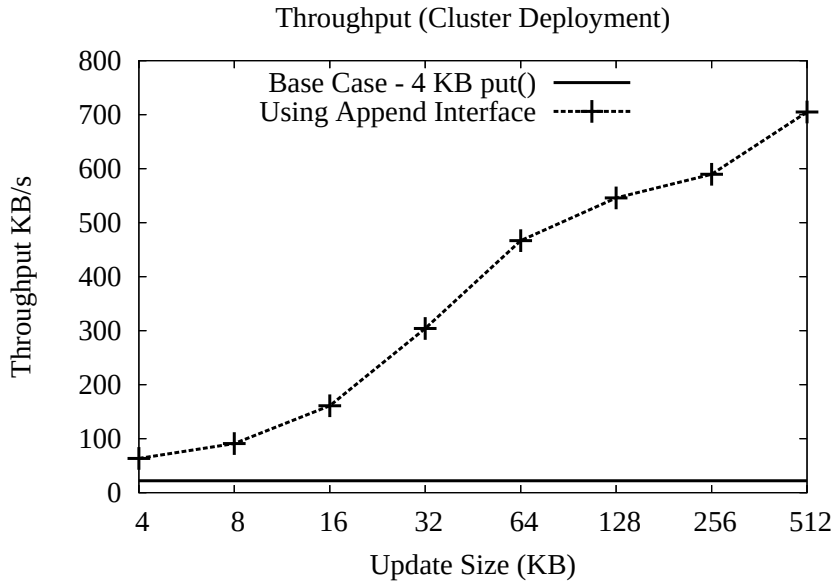


Figure 11.1: Aggregation increases system throughput by reducing computation at the data source and in the infrastructure. The base case shows the throughput of a client that stores 4 KB blocks (and a certificate) using `put()` operation, as in a traditional DHT.

time is often greater then 210.5 and 10.8 ms.

We apply load to these deployments using 32 nodes of a different local cluster. Each machine in the *test cluster* has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory, and two 36 GB disks. Signature creation and verification takes an average of 6.0 and 0.6 ms, respectively. The cluster shares a 100 Mbps link to the external network. This cluster is also a shared site resource, but its utilization is lower than the storage cluster.

11.2 Cluster Deployment

We first report on results from a deployment of Antiquity on the local storage cluster. In addition to serving as a tool for testing and debugging, this deployment also allows us to observe the behavior of the system when bandwidth is plentiful and contention for the processor is relatively low.

Figure 11.1 shows how aggregation improves performance. In this test, a single data source submits synchronous updates of various sizes to Antiquity. At the data source, aggregation reduces the cost of interacting with the system by amortizing the cost of creating and signing

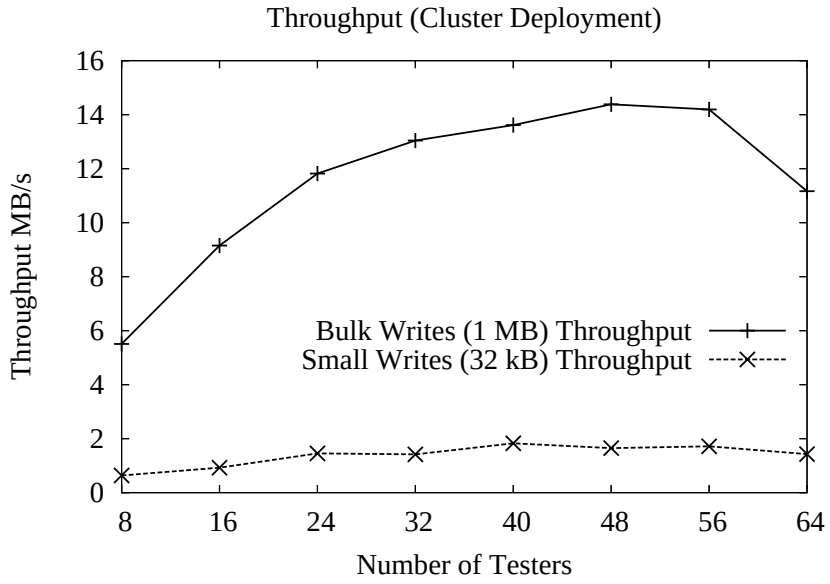


Figure 11.2: The throughput of the system scales with the number of users until resources at the storage servers are saturated. Performing bulk writes using the `put()` interface, the cluster deployment becomes saturated with 48 data sources. Using the `append()` interface, the sustained throughput is much lower because each quorum operation adds only a small amount of data to the log.

certificates and transmitting network messages over more data. In the storage system, aggregation reduces the number of quorum operations that must be performed to write a given amount of data to the system. For comparison, we show the throughput of a data source that stores data using synchronous `put()` operations with payload of 4 KB of application data, as in typical in a DHT. The `put()` throughput is lower than `append()` operations of equivalent size because `put()` operations require a new configuration be created by the administrator.

Figure 11.2 shows how the throughput of the system scales with load. In this test, multiple data sources, each with a distinct key pair and log, submit synchronous updates to the system. In some tests, the source writes data to the log incrementally in 32 KB chunks using the `append()`/`snapshot()`/`truncate()` interface; in other tests, the source writes data to the log in bulk using the `put()` operation. In all tests, extents have a maximum capacity of 1 MB. The graph shows that, using the `put()` interface, throughput increases with the number of users up to 48 users. With additional data sources, contention at the storage servers reduces throughput. We would expect this number to increase if we could spread the load across more servers and network links. The throughput of sources using the `append()` interface is substantially lower because each quorum operation adds a relatively small amount of data to the log.

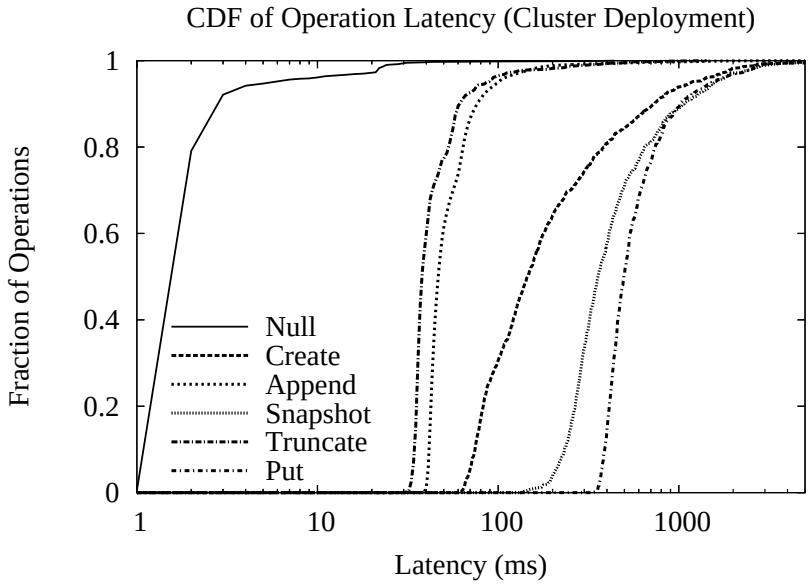


Figure 11.3: Different operations have widely varying latency. The latency is dependent on the amount of data that must be transferred across the network and the amount of communication with the administrator required. The latency CDF of all operations (even the `null()` RPC operation) exhibits a long tail due to load from other, unrelated jobs running on the shared cluster.

Next, we measure the latency of individual operations. In this test, a single data source issues a variety of operations, including incremental writes of 32 KB using the `append()` interface and bulk writes using the `put()` interface. Extents are configured to have a maximum capacity of 1 MB. Figure 11.3 presents a CDF of the latency of various operations. Table 11.1 presents a breakdown of the median latency times. The latency of different types of operations various significantly. The `append()` and `truncate()` operations are the fastest because they transfer little or no data and do not require any interaction with the administrator. The `create()` operation is slightly slower because, though it contains no application payload, it must contact the administrator to obtain a new configuration. Finally, the `snapshot()` and `put()` operations are the slowest; they transfer large amounts of data and must contact the administrator to find a suitable configuration of storage servers. The latency distribution of all operations exhibit a long tail due to load from other unrelated processes running on the same machines; note, even the `null()` RPC call can take longer than one second. This delay is due to load from unrelated jobs running on the shared cluster.

Table 11.1 illustrates that interacting with the DHT consumes a significant fraction of time. In particular, `append()` and `truncate()` interact with the DHT one time to `publish()` the soundness proof, although this is necessary for repair. However, operations that create extents

	Time (ms)				
	No Admin		Admin		
Phase	`truncate()`	32 kB `append()`	`create()`	`snapshot()`	1 MB `put()`
T_{req}					
Signs Request	6.0	6.0	6.0	6.0	6.0
Send Request	1.8	4.2	1.8	1.8	81.6
Verify Request	0.6	1.0	0.6	0.6	13.1
DHT `lookup()` Locations	(cached) 0.0	(cached) 0.0	13.2	13.2	13.2
DHT `publish()` Coordinator	(cached) 0.0	(cached) 0.0	7.2	7.2	7.2
subtotal	8.4	11.2	28.8	28.8	121.1
T_{create_config}					
DHT `lookup()` Neighbors			6.6	6.6	6.6
Send Config Request			1.6	1.6	1.6
Verify Config Request			0.6	0.6	0.6
Create New Config			8.2	8.2	8.2
Sign New Config			3.2	3.2	3.2
Reply w/ New Config			1.6	1.6	1.6
subtotal	0.0	0.0	21.8	21.8	21.8
T_{quorum}					
Send Request	1.8	6.6	1.8	1.8	157.4
Verify Request	0.6	1.0	1.2	1.2	13.7
Fetch Extent				98.4	
Disk	4.1	5.9	4.1	61.9	61.9
DHT `publish()` Location			7.2	62.3	42.3
Sign Result	3.2	3.2	3.2	3.2	3.2
Send Reply	1.6	1.6	1.6	1.6	1.6
Verify Replies	4.2	4.2	4.2	4.2	4.2
DHT `publish()` Proof	7.2	7.2	63.3	63.3	63.3
subtotal	22.7	29.7	86.6	297.9	347.6
T_{resp}					
Reply w/ Proof	1.7	1.7	1.7	1.7	1.7
Verify Proof	4.2	4.2	4.2	4.2	4.2
subtotal	5.9	5.9	5.9	5.9	5.9
Total – Median (Min)	37.0 (31.0)	46.8 (38.0)	143.1 (62.0)	354.4 (137.0)	496.4 (338.0)

Table 11.1: Measured breakdown of the median latency times for all operations. For all operations, the client resides in the test cluster and the administrator and storage servers reside in the storage cluster. The average network latency and bandwidth between applications on the test cluster and storage cluster is 1.7 ms and 12.5 MB/s (100 Mbs), respectively. The average latency and bandwidth between applications within the storage cluster is 1.6 ms and 45.0 MB/s (360 Mbs). All data is stored to disk on the storage cluster using BerkeleyDB which has an average latency and bandwidth of 4.1 ms and 17.3 MB/s, respectively. Signature creation/verification takes an average of 6.0/0.6 ms on the test cluster and 3.2/0.6 ms on the storage cluster. Bandwidth of the SHA-1 routine on the storage cluster is 80.0 MB/s. Finally, DHT `lookup()` and DHT `publish()` take an average of 4.2 ms and 7.2 ms, respectively.

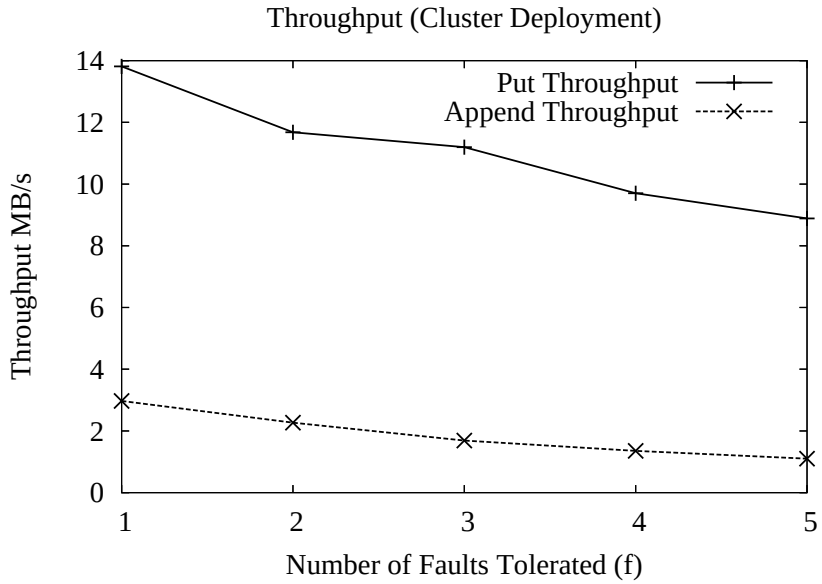


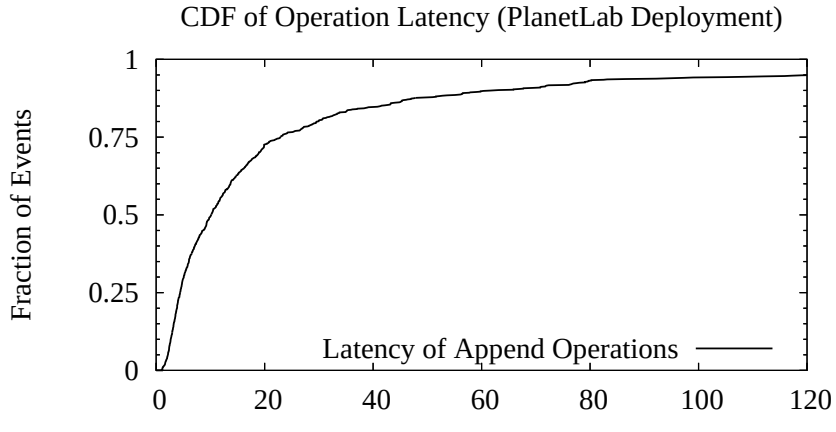
Figure 11.4: Increasing the deployment's tolerance to faults reduces the system throughput since the system must transfer more data with each write operation.

(`create()`, `snapshot()`, `put()`) interact with the DHT over five times. Furthermore, multiple DHT `publish()` operations to the same identifier often take longer than expected since (locally to a DHT server) `publish()` sometimes competes with other BerkeleyDB operations for use of the disk (e.g. BerkeleyDB log cleaning).

Figure 11.4 shows how the size of a configuration affects system throughput. In this experiment, we vary the deployment to tolerate varying number of faults, f, in a configuration. For each arrangement, the size of a configuration is $3f+1$. When the size of the configuration increases, the system must transfer more data with each write operation. The extra messages and bandwidth reduce write throughput. As expected, because the quorum protocols requires $O(n)$ messages, the throughput roughly decreases linearly with the number of faults tolerated. We would expect the throughput to drop more quickly for designs based on Byzantine agreement because those protocols require $O(n^2)$ communication.

11.3 PlanetLab Deployment

Next, we report on results from a deployment of Antiquity on PlanetLab. For reasons illustrated in Figure 11.5, the focus of our evaluation of the PlanetLab deployment is not on its



Min	25%	50%	90%	95%	99%	Max
1.08	4.41	10.2	63.1	124.5	302.1	615.9

Figure 11.5: The latency of operations on PlanetLab varies widely depending on the membership and load of a configuration. As an example, this graphs plots the CDF of the latency for appending 32 KB to logs stored in the system. The table highlights key points in the curves.

performance but data maintenance over time. That graph plots the CDF of the latency of more than 800 operations that append 32 KB of data to logs in the systems. The accompanying table reports several key points on the curve. Given the best of circumstances, the latency of an `append()` operation can be as low as one second. However, when configurations include distant or overloaded servers or bandwidth is restricted on some path, the latency increases considerably. Because of the characteristics of the PlanetLab testbed, many operations are very slow.

Our deployment of PlanetLab focuses on how the design maintains data over time, especially as machines fail. We built a simple test application that writes logs to the system and periodically checks that they are still available. Each log consists of one key-verified extent (the log head) and an average of four hash-verified extents (the number of hash-verified extents vary uniformly with an average of four). Key-verified extents vary in size uniformly up to 1 MB; all hash-verified extents are 1 MB. The average size of a log is 4.5 MB (0.5 MB log head and 4 x 1MB hash-verified extents). The test application stores 18,779 logs (18,779 log heads and 75,085 hash-verified extents) totaling 84 GB. After writing an extent to the system, this test application records a summary of the extent in a local database. No data was lost, even though 10% of the servers suffered permanent failures.

We perform various tests to measure the efficacy of the Antiquity deployment. First, we

measure the percent of extents with at least a quorum of replicas available and in a consistent state in Section 11.3.1. This test is important since it measures the percent of extents where progress can be made. Next, In Section 11.3.2, we measure the cost of maintaining secure logs in terms of replicas created. In particular, we measure the average number of replicas created per unit time and the total number of replicas created. This test measures the systems ability to maintain sufficient replication levels in response to server failure.

11.3.1 Quorum Consistency and Availability

In this section we measure the quorum consistency and availability. Our first experiment uses an application to periodically read an extent. Every 10 seconds, it selects at random an entry from the database and attempts to contact a quorum of the servers hosting that extent. It reports whether it was able to reach a quorum of servers. It also verifies that the replicas are in a consistent state and that state matches what was written. Our second measurement uses a server availability trace, server database log, and extent configuration to produce a similar consistency and availability metric. The first experiment measures the expected application performance and includes intermittent effects such as server load, network performance, etc.; whereas, the second measurements ignore such effects and simply use server availability.

Figure 11.6.(a) shows the results of this first test. The x-axis shows the two month duration of the test. The top curve shows the percentage of successful quorum checks. A software bug between May 13 through 23 caused over half the servers not to respond to RPC requests. Periodic server application reboot temporarily masked the bug. But the performance continued to degrade until the problem was solved on May 23. Over the life of the test (including the May 13 through 23 interruption), 94% of checks reported that a quorum of servers was reachable and stored a consistent state of the extent. This figure matches a computed estimate for the number of valid configuration. Using a monitor on the remote hosts, We have measured the average availability of machines in PlanetLab to be 90%. Note, this figure indicates that the node is up, not necessarily that the node can be reached over the network. Given that measurement, we would expect a quorum of 5 (out of 7) servers to be available 94% of the time.

The lower curve on the plot shows the percentage of checks that failed due to RPC failures, network disruptions, and other timeouts. We attempt to reach a quorum through 5 different gateways before marking a check failed. Our measurements show that up to 90% of the failed checks may be caused by components outside of Antiquity. This percentage increases as the load on PlanetLab

increases. Furthermore, the high load causes a number of Antiquity processes to be terminated due to resource exhaustion. Thus, the actual percentage of consistent quorums (shown next) is higher than the 94% measured from the application.

Figure 11.6.(b) shows the results of the second test. The x-axis shows the two month duration of the test. The top curve shows the percentage of extents where a quorum is available and consistent as measured by a server availability trace, database log, and extent configuration. The server availability trace ignores the software bug; as a result, until May 23, 100% of the extents had at least a quorum available and consistent. After May 23, however, server churn increased, tripling from 24 server failures per hour to 76. The cause for the increase in server churn is a watchdog timer that restarts a server's Antiquity application when it is unresponsive for over six minutes. Figure 11.7 shows the number of servers available and server failures during each hour of the test.

11.3.2 Quorum Repair

Antiquity's repair process is critical to maintain the availability of a quorum of servers for each extent. Figure 11.8 plots the cumulative number of replicas created in the PlanetLab deployment. During the period of observation, Antiquity initially created a total of 657,048 extent replicas (each of the 93,864 extents were initially created with 7 replicas). The replicas initially accounted for 577 GB of replicated storage (84 GB of unique storage).

In order to maintain the availability of a quorum of servers, Antiquity triggers a quorum `repair()` protocol when less than a quorum of replicas are available. Each `repair()` replaces at least three replicas since that is the least number of unavailable servers required to trigger `repair()` with $f = 2$. The deployment experienced an average of 114 failures per hour (Antiquity application failures). In response to failures, Antiquity triggered `repair()` 92 times per hour. As the number of unavailable servers accumulated, nearly every failure triggered a `repair()`. Each `repair()` replaced an average of four replicas. As a result, Antiquity created a total of 653,028 replicas due to `repair()` during the two month period of observation which cost the system less than 0.31 KB/s (320 Bps) per server due to `repair()`. Coupled with maintaining the availability and consistency of up to 97% of the extents, this demonstrates that Antiquity is capable of maintaining sufficient replication levels in response to server failure.

11.4 A Versioning Back-up Application

Finally, we have built a versioning back-up application that stores data in Antiquity. The application translates a local file system into a Merkle tree as shown in Figure 6.4 and used in similar previous systems [MT85, DKK+01]. The application records in a local database when data was written to the infrastructure. It checks the local database before archiving any new data. This acts as a form of copy-on-write, reducing the amount of data transmitted.

We stored the file system containing the Antiquity prototype (source code, object code, utility scripts, etc.) in PlanetLab. The file system is recorded in 15 1-MB extents. The system has repaired two of the 15 extents while ensuring both consistency and durability of the file system.

11.5 Experience and Discussion

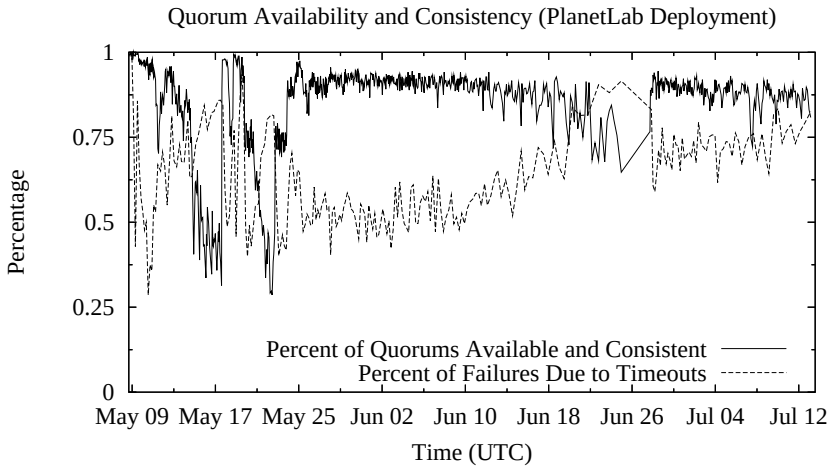
Putting it all together, Antiquity maintained 100% durability and 97% quorum availability of 18,779 logs broken into 93,864 extents. On average, when a server failed, it took the system 30 minutes to detect and classify the server as failed (value of timeout) and three hours to replace replicas stored on failed server with less than a quorum of remaining replicas available. Once repair completed for a particular extent, at least a quorum of servers was again available. Reflecting on our experience, the structure of the secure log made this an easier task for three reasons.

First, maintaining the integrity of a secure log is easier than other structures since the verifier for the log (and each extent) defines the order of appends and cryptographically ensures the content. In particular, there is only one sequence of appends that results in a particular verifier. This verifier is used as a predicate to ensure that new writes are appended to the log in a consistent fashion. Furthermore, this verifier is used by the storage system to ensure that each replica stores the same state. In the deployment, this verifier was a critical component used to ensure the consistency and integrity of the log and all of its extents. Furthermore, it is cheap to compute, update, and compare.

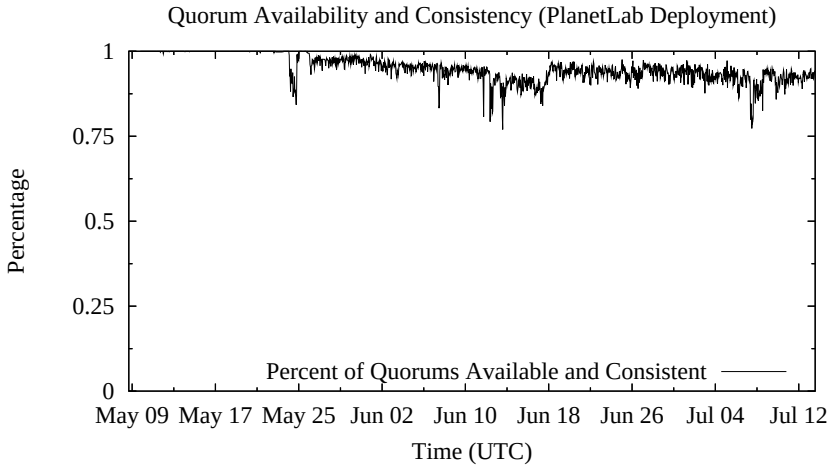
Second, a storage system that implements a secure log is a layer or middleware in a larger system. The secure log abstraction bridges the storage system and higher level applications together. In fact, the secure log interface implemented by Antiquity is a result of breaking OceanStore into layers. In particular, a component of OceanStore was a primary replica implemented as a Byzantine Agreement process. This primary replica serialized and cryptographically signed all updates. Given this total order of all updates, the question was how to durably store and maintain the order? Fur-

thermore, what should be the interface to this storage system? An append-only secure log answered both questions. The secure log structure assists the storage system in durably maintaining the order over time. The append-only interface allows a client to consistently add more data to the storage system over time. Finally, when data is read from the storage system at a later time, the interface and protocols ensure that data will be returned and that returned data is the same as stored.

Finally, self-verifying structures such as a secure log lend themselves well to distributed repair techniques. The integrity of a replica can be checked locally or in a distributed fashion. In particular, we implemented a quorum repair protocol where the storage server replicas used the self-verifying structure. The structure and protocol provided proof of the contents of the latest replicated state and ensured that the state was copied to a new configuration.



(a) Periodic Application Read



(b) Server Availability Trace

Figure 11.6: Quorum Consistency and Availability. (a) Periodic reads show that 94% of quorums were reachable and in a consistent state. Up to 90% of failed checks are due to network errors and timeouts. (b) Server availability trace shows that 97% of quorums were reachable and in a consistent state. This illustrates the increase in performance over (a) where timeouts reduced the percent of measured available quorums.

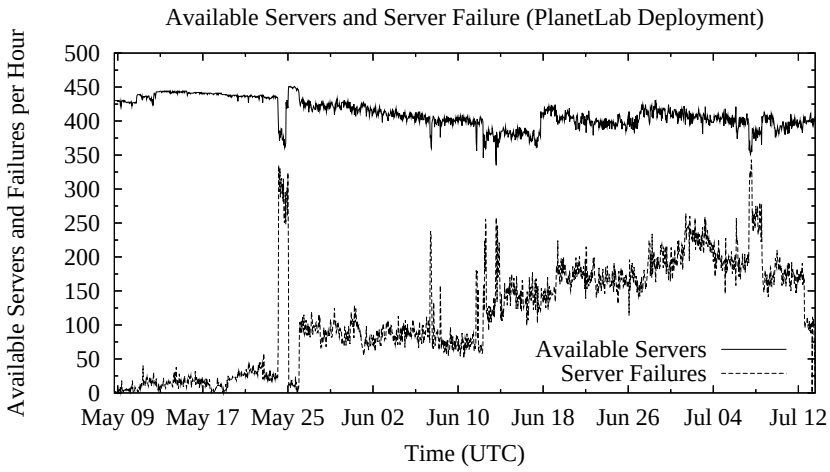


Figure 11.7: Number of servers with their Antiquity application available per hour. Additionally, number of failures per hour. Most failures are due to restarting the unresponsive Antiquity instances. As a result, a single server may restart its Antiquity application multiple times per hour if the instance is unresponsive.

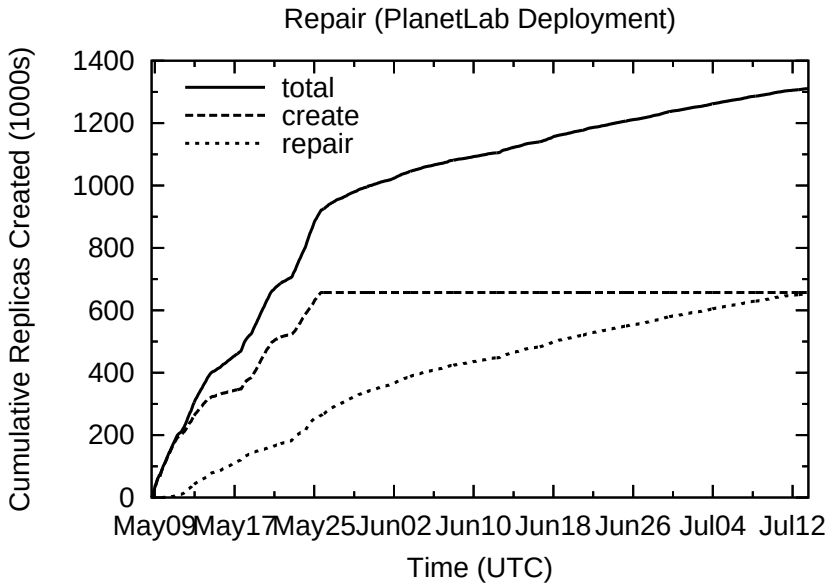


Figure 11.8: Number of replicas created over time due to storing new data and in response to failure.

Part V

Related and Future Work

Chapter 12

Related Work

This thesis focuses on wide-area distributed storage systems. It benefits from prior analyses and experience of many prior systems. Table 12.1 illustrates a portion of the design space of many systems that utilize replication. We discuss the systems and analyses further.

12.1 Logs

The log-structured file system (LFS) [MRC+97] used a log abstraction to improve the performance of local file systems. Schneier and Kelsey [SK98] and SUNDR [LKMS04] demonstrated how to use a secure log to store data on an untrusted remote machines. They do not address how to replicate the log.

12.2 Byzantine Fault-Tolerant Services

Byzantine fault-tolerant services have been proposed to help meet the challenges of unsecured, distributed environments. FarSite [ABC+02], OceanStore [REG+03], and Rosebud [RL03] aim to build distributed storage systems using Byzantine fault-tolerant agreement protocols [LSP82, CL99]. Abd-El-Malek et al [AGG+05], Goodson et al [GWGR03], the COCA project [ZSv00], Fleet [MRTZ01], and Martin and Alvisi [MA04] build reliable services using Byzantine fault-tolerant quorum protocols [MR97]. HQ Replication combines both Byzantine fault-tolerant agreement and quorum protocols to reduce communication for the common case and order conflicting updates. Martin and Alvisi define a protocol that allows the configuration to be changed with the

System	Byzantine tolerance	wide-area	mutable	security	durability	consistency	aggregation	update efficiency	maintenance efficiency
Antiquity	Y	Y	Y	Y	Y	Y	Y	Y	Y
Pond [REG+03]	Y	Y	Y	Y	Y	Y	N	N	N
Byzantine Agreement [LSP82, Sch90] Castro/Liskov [CL99]	Y	?	Y	Y	Y	Y	N	N	?
HQ Replication [CML+06]	Y	?	Y	Y	Y	Y	N	Y	?
Byzantine Quorums [MR97] CMU [AGG+05, GWGR03]	Y	?	Y	Y	Y	Y	N	Y	N
Dynamic Byzantine Quorums [MA04]	Y	?	Y	Y	Y	Y	N	Y	Y
COCA [ZSv00]	Y	?	?	Y	Y	Y	?	Y	?
Chain Replication [vS04]	N	N	Y	N	Y	Y	N	Y	Y
Secure Log [SK98, CPH01]	N	N	Y	Y	?	Y	?	Y	N/A
Venti [QD02]	N	N	Y	?	Y	?	N	Y	N/A
Rosebud [RL03]	Y	?	Y	?	Y	Y	N	N	?
Etna [MGM04]	N	Y	Y	N	Y	Y	N	Y	?
Carbonite [CDH+06]	N	Y	N	N/A	Y	N/A	N	N/A	Y
Glacier [HMD05]	N	Y	N	N/A	Y	N/A	Y	N/A	?
OpenDHT [RGK+05] Dhash [Cat03] PAST [DR01]	N	Y	?	N	Y	N	N	?	?
TotalRecall [BTC+04]	N	Y	?	N	Y	?	N	?	Y
Myriad [LMPZ02], EMC [Cor], Distributed DBs [DGH+87] (i.e. Mirroring)	N	?	Y	?	Y	?	N	Y	?
GFS [GGL03]	N	?	?	N	?	?	Y	Y	Y
RAID [PGK88] Harp [LGG+91] Petal [LT96]	N	N	Y	N	?	?	N	Y	Y

Table 12.1: System Comparison

help of an administrator. None of the agreement or quorum systems reactively trigger reconfiguration.

12.3 Wide-area Distributed Storage Systems

Many researchers have used distributed hash table (DHT) technology to build wide-area distributed storage systems. Notable examples are Carbonite [CDH+06], CFS [DKK+01], Glacier [HMD05], Ivy [MMGC02],PAST [DR01], Total Recall [BTC+04], and Venti [QD02]. Carbonite and Total Recall optimize for the wide-area by reducing the number of replicas created due to transient failures. Glacier uses aggregation to reduce storage overheads. Ivy uses a log structure similar to Antiquity; however, the log is block-based instead of extent-based. None of these systems implement a Byzantine consistency algorithm. Chain Replication [vS04] and Etna [MGM04] both implement consistency protocols, but assume fail-stop failures.

12.4 Replicated Systems

Replicated systems like GFS [GGL03], Harp [LGG+91], Petal [LT96], RAID [PGK88], and XFS [ADN+95] have been shown to reduce the risk of data loss. GFS and XFS also use aggregation. These systems target well-connected environments.

Distributed databases [DGH+87], distributed operating systems such as Amoeba [Tvv+90], online disaster recovery systems such as Myriad [LMPZ02], and EMC storage systems [Cor] use the wide-area replication to increase durability. Myriad and EMC replicate data between a primary and backup site. Wide-area recovery is initiated after site failure; single disk failure is repaired locally with RAID.

12.5 Replication analysis

The use of a birth-death data-loss model is a departure from previous analyses of reliability. Most DHT evaluations consider whether data would survive a single event involving the failure of many nodes [DKK+01, BR03, WK02]. This approach does not separate durability from availability, and does not consider the continuous bandwidth consumed by replacing replicas lost to disk failure.

Fault-tolerant storage systems designed for single-site clusters typically aim to continue operating despite some fixed number of failures. These systems often choose the number of replicas with an eye to a voting algorithm to ensure correct updates in the presence of partitions or Byzantine failures [LGG+91, CL02, GBHC00, SFV+04, LS00].

Our birth-death model is a generalization of the calculations that predict the MTBF for RAID storage systems [PGK88]. Owing to its scale, a distributed system has more flexibility to choose parameters such as the replication level and number of replica sets when compared to RAID systems.

Blake and Rodrigues argue that wide-area storage systems built on unreliable nodes cannot store a large amount of data [BR03]. Their analysis is based on the amount of data that a host can copy during its lifetime and mirrors our discussion of feasibility. We come to a very different conclusion because we are considering a stable system membership where data loss is driven by disk failure. Blake and Rodrigues assumed a system with continual membership turnover.

FAB [SFV+04] and Chain Replication [vS04] both consider how the number of possible replica sets affects data durability. The two come to opposite conclusions: FAB recommends a small number of replica sets since more replica sets provide more ways for data to fail. Chain replication recommends many replica sets to increase repair parallelism and thus reduce repair time. These observations are both correct: choosing a replica placement strategy requires balancing the probability of losing some data item during a simultaneous failure (by limiting the number of replica sets) and improving the ability of the system to tolerate a higher average failure rate (by increasing

the number of replica sets and reconstruction parallelism).

Nath et al [NYGS06] demonstrates that most correlated failures are small, involve few servers, and are predictable given the failure of a server (e.g. within the same site). Further, most of these small correlated failure events do not cause data loss since they likely do not destroy all the replicas for a particular objects. However, large correlated failure events that cause many servers to fail simultaneously occur very infrequently and are unpredictable [NYGS06]. As a result, models used to estimate the number of replicas to create should consider correlated failures; however, deployed systems that use simple replica placement strategies such as random (with small optimizations) are often sufficient to avoid most observed correlated failures.

12.6 Replicated systems

Petal [LT96], DDS [GBHC00], Map Reduce [DG04], xFS [ADN+95], and Harp [LGG+91] all employ replication to deal with failures. They are designed for LANs where bandwidth is plentiful and transient failures are rare. As a result they can maintain a small, fixed number of replicas and create a new replica immediately following a failure.

Distributed databases [DGH+87] use wide-area replication to distribute load and increase durability. These systems store mutable data and focus on the cost of propagating updates, a consideration not applicable to the immutable data we assume.

Total Recall is the system most similar to our work [BTC+04]. We borrow from Total Recall the idea that creating and tracking additional replicas can reduce the cost of transient failures. Total Recall's lazy replication keeps a fixed number of replicas and fails to reincorporate replicas that return after a transient failure if a repair had been performed. Total Recall must also use introspection or guessing to determine an appropriate high water mark that Carbonite can arrive at naturally.

Glacier [HMD05] focuses on durability despite correlated failures, while we aim to withstand only those bursts of failures that one would ordinarily expect with random uncorrelated failures.

Beehive [RS04] creates and places replicas of objects to meet a target lookup latency. The techniques in this paper use replication to provide only durability; we rely on routing optimizations to reduce latency [DLS+04].

Our systems store data in the DHT; an alternative is to store data on designated storage servers and use the DHT to store pointers to those nodes [REG+03, K+00]. This arrangement

simplifies replica maintenance since much less data needs to be maintained in the DHT.

Many systems use mirroring to maintain data durably in the wide area [Cor, PMF+02]. Data is replicated between a primary and backup sites and further replicated locally at each site using RAID. Wide area recovery is initiated only after site failure; individual disk failure can be repaired locally. The techniques presented in this paper are relevant since the amount of data to be transferred after a failure is large compared to the wide area network link capacities.

12.7 Digital Libraries

Digital libraries such as LOCKSS [MRG+05] preserve journals and other electronic documents for significantly long periods of time. Durability of documents is the primary goal and availability is secondary. The documents are read-only and cannot be updated. The documents are replicated at many sites to maintain durability. Many of the documents stored do not have an "owner"; as a result, the system relies on voting to maintain the integrity. This design is different than a distributed wide-area on-line archival storage system where there is an owner for each document. Furthermore, the document can be modified and the system ensures that the stored state reflects changes made by the owner.

Chapter 13

Future Work

In this chapter, we revisit the assumptions and limitations of the approach described in previous chapters, discussing opportunities for future work that we hope to pursue in the future.

13.1 Proactive Replication for Data Durability

Wide-area storage systems replicate data for durability. A common way of maintaining the replicas is to detect server failures and respond by creating additional copies. This reactive technique can minimize total bytes sent since it only creates replicas as needed. However, it can create spikes in network use after a failure. These spikes may overwhelm application traffic and can make it difficult to provision bandwidth.

Most existing distributed wide-area storage systems use a reactive technique to maintain data durability [BTC+04, CDH+06, DLS+04, HMD05, RGK+05]. The bandwidth needed to support this reactive approach can be high and bursty: each time a server fails permanently, the system must quickly produce a new copy of all the objects that the server had stored [BR03]. Quick replication is especially important for storage intensive applications like OceanStore/Pond [REG+03], OverCite [SLC+06], or ePOST [MPHD06] where data loss must be minimized. While reactive systems can be tuned to provide durability at low total cost [CDH+06], the need to repair quickly can cause dramatic spikes in bandwidth use when responding to failures. In many settings, provisioning for high peak usage can be expensive.

Proactively replicating objects before failures occur is an alternative to maintaining data durability. In particular, proactive replication constantly creates additional redundancy at low rates. This technique evens out burstiness in maintenance traffic by shifting the time at which bandwidth

is used. Instead of responding to failures, a proactive maintenance system operates constantly in the background, increasing replication levels during idle periods. Operating proactively in this manner results in a predictable bandwidth load: server operators and network administrators need not worry that a sudden burst of failures will lead to a corresponding burst in bandwidth usage that might overwhelm the network. Instead, any burstiness in network usage will be driven by the application's actual workload. The question is whether this method can still prevent data from being lost.

Tempo [SHD+06] proposed by Sit et al. is a proactive maintenance scheme. In contrast to systems that use as much bandwidth as necessary to meet a durability specification (given explicitly [BTC+04] or in the form of a minimum replication level [DLS+04]), each server in Tempo, a proactive replication system, operates under a bandwidth budget specified by the server operator. A budget is attractive because it is easy for the user to configure: bandwidth is a known, measurable and easily understood quantity. The servers cooperate and attempt to maximize data durability within their individual budgets by constantly creating new replicas, whether or not they are needed at the moment. While systems that specify a number of replicas respond to failures by varying the bandwidth usage in an attempt to maintain that replication level, proactive replication systems instead effectively adjusts the available replication level subject to its bandwidth budget constraints. Tempo showed that in a simulation based on PlanetLab measurements over a 40 week period, proactive replication can maintain more than 99.8% of a 1TB workload durably using as little as 512 bytes per second of bandwidth on each server. With 2K per second per server, no objects were lost: this amount of bandwidth is comparable to that used by reactive systems but proactive replication uses this much more evenly.

Proactive replication with constant repair traffic is an interesting concept and deserves further investigation. For instance, it could be used in domains where maintaining a constant data repair rate might be as important as maintaining the minimum. For example, a storage system with proactive repair can better ensure quality of service (e.g. response time) observed by applications by dedicating a specified amount of the bandwidth budget for repair. Sensor networks that monitor and store samples of their environment are another example. Proactive repair would allow designers to better calculate expected lifetime of the sensor network based on expected power usage due to proactive repair.

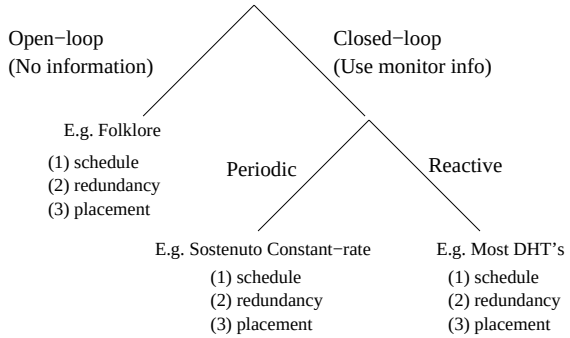


Figure 13.1: Design Space for Repair Algorithms.

13.2 Closed-loop, Proactive Repair, for Data Durability

Both reactive and proactive repair schemes discussed are closed-loop systems, they monitor the number of available replicas in order to decide which data object to repair. In particular, closed-loop systems, sense and respond to the current state of a system. For example many systems monitor server availability and keep track of the set of replicas that are stored on each server. Once repair is initiated, the storage system can make an informed decision on the number of new replicas to create.

Open-loop systems are an alternative. An open-loop storage system does not use any monitored information. As a result, an open-loop system periodically creates new replicas regardless of the number of available replicas and whether any replicas have actually failed or not. The simple analysis in Section 4.1.1 is an example of an open-loop system.

Servers in open-loop systems independently decides when to create an additional replica for a data object and where to store the new replica. This "state-less" form of replication is potentially easier to design and implement. However, to ensure some level of data durability, an open-loop system uses the most resources of any algorithm proposed thus far. Figure 13.1 shows the design space. Nevertheless, this state-less form of replication may be ideal in environments where communication required for monitoring and coordination is expensive. Such environments include sensor networks. Open-loop storage systems could be used to durably store data with an expected and constant communication cost.

13.3 Administrator Discussion

The dynamic Byzantine quorum protocol presented in Part III and evaluated in Part IV relies on an administrator which is a potential vulnerability. The administrator is assumed to be trusted, non-faulty, and always available. As a result, a violation of these assumptions could threaten data durability or integrity.

There are three attacks a faulty or malicious administrator can perform. First, a faulty administrator may create configurations with many faulty storage servers for a data object. Such configurations could ignore protocol preventing the client from making changes to the data or preventing data from being stored durably. Second, a faulty administrator may be unavailable for long periods of time preventing necessary configuration changes from occurring. Configuration changes are necessary to maintain data durability since a new configuration is required to replace failed servers. Finally, a faulty administrator may assign disjoint configurations with the same configuration sequence number. Disjoint configurations could create a fork consistency [LKMS04], disjoint data modification paths where the client is unaware of the multiple paths.

Creating an *administrator replicated service* can reduce or eliminate the above attacks. Replicating the administrator into independent processes and requiring the administrator replicas to agree to authorize and sign configurations would make the administrator appear to be a fail-stop entity, either follows protocol correctly or produces no results. The administrator replicated service could tolerate up to f faulty or failed administrator replicas and produce correct results. It would produce no results with more than f faulty or failed replicas. The system requires human intervention with more than f administrator replica failures since the service could not uphold any guarantees.

13.4 OceanStore as an Application

The secure log interface is a primitive and its implementation is a component in a larger system. Because of its simplicity, narrow interface, and provable properties, storage systems that implement a secure log interface such as Antiquity can be built and deployed. The next phase is to build a complete system layered on top of such a storage system. An interesting client of the storage system could be OceanStore. OceanStore could utilize a storage system like Antiquity as a storage service.

OceanStore can use the secure log interface to ensure that all updates are stored durably

and the order of updates are correctly maintained over time. The secure log interface and append-only usage model is a good fit since OceanStore already serializes and cryptographically signs all updates via a primary replica. The primary replica appears to the storage system as a fail-stop client since it is implemented as a Byzantine Agreement process. Furthermore, OceanStore clients can use the secure log interface to ensure data read from the storage system at a later time will be returned and that returned data is the same as stored.

A storage system such as Antiquity can benefit other components of OceanStore as well. In particular, OceanStore aggressively caches data via secondary replicas. Instead of disseminating updates to secondary replicas, OceanStore can send "notifications" of updates and secondary replicas can pull new updates from the storage system as needed.

13.5 Summary

In this chapter, we summarized some of the more interesting and challenging areas of future work related to distributed wide-area on-line archival storage systems. Of particular importance are notions of repair because repair affects the efficiency and correctness of the storage system. Critical is deciding when to trigger repair, what information should be used, and other requirements such as a signature from an administrator. Alternatively, we suggest that maybe the minimum total cost of repair is not the most important consideration, rather a constant rate might be better. With a constant repair rate, storage systems could give better quality of service guarantees to applications, could be deployed in other environments such as sensor networks, and could lend themselves to simpler systems to build and analyze. These diverse problems illustrate many unexplored issues related to distributed wide-area on-line archival storage – an indicator of future research directions.

Chapter 14

Concluding Remarks

As the amount of digital assets increase, systems that ensure the durability, integrity, and accessibility of digital data become increasingly important. Distributed on-line archival storage systems are designed for this very purpose. This thesis explored several important challenges pertaining to fault tolerance, repair, and integrity that must be addressed to build such systems.

The first part of this thesis explored how to maintain durability via fault tolerance and repair and presents many insights on how to do so efficiently. Fault tolerance ensures that data is not lost due to server failure. Replication is the canonical solution for data fault tolerance. The challenge is knowing how many replicas to create and where to store them. Fault tolerance alone, however, is not sufficient to prevent data loss as the last replica will eventually fail. Thus, repair is required to replace replicas lost to failure. The system must monitor and detect server failure and create replicas in response. The problem is that not all server failure results in loss of data and the system can be tricked into creating replicas unnecessarily. The challenge is knowing when to create replicas. Both fault tolerance and repair are required to prevent the last replica from being lost, hence, maintain data durability.

The second part of this thesis explored how to ensure the integrity of data. Integrity ensures that the state of data stored in the system always reflects changes made by the owner. It includes non-repudiably binding owner to data and ensuring that only the owner can modify data, returned data is the same as stored, and the last write is returned in subsequent reads. The challenge is efficiency since requiring cryptography and consistency in the wide-area can easily be prohibitive.

Next, we exploited a secure log to efficiently ensure integrity. We demonstrate how the narrow interface of a secure, append-only log simplifies the design of distributed wide-area storage systems. The system inherits the security and integrity properties of the log. We describe how

to replicate the log for increased durability while ensuring consistency among the replicas. We present a repair algorithm that maintains sufficient replication levels as machines fail. Finally, the design uses aggregation to improve efficiency. Although simple, this interface is powerful enough to implement a variety of interesting applications.

Finally, we applied the insights and architecture to a Prototype called Antiquity. Antiquity efficiently maintains the durability and integrity of data. It has been running in the wide area on 400+ PlanetLab servers where we maintain the consistency, durability, and integrity of nearly 20,000 logs totaling more than 84 GB of data despite the constant churn of servers (a quarter of the servers experience a failure every hour)..

Bibliography

[ABC+02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, December 2002.

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[ADN+95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proc. of ACM SOSP*, December 1995.

[Age] National Security Agency. Global information grid (gig). `http://www.nsa.gov/ia/industry/gig.cfm`. Last accessed September 2006.

[AGG+05] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. of ACM SOSP*, October 2005.

[AHK+02] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proc. of USENIX FAST*, January 2002.

[And04] D. Andersen. *Improving End-to-End Availability Using Overlay Networks*. PhD thesis, Massachusetts Institute of Technology, 2004.

[BBC+04] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, , and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, March 2004.

[BDET00] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of Sigmetrics*, June 2000.

[BKK+95] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[BR03] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of HOTOS*, May 2003.

[BSV03] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. of IPTPS*, February 2003.

[BTC+04] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Totalrecall: Systems support for automated availability management. In *Proc. of NSDI*, March 2004.

[Cat03] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, MIT, June 2003.

[CDH+06] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, San Jose, CA, May 2006.

[CEG+96] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE*, pages 485–495, February 1996.

[CL99] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.

[CL00] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. of OSDI*, 2000.

[CL02] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[CML+06] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. of OSDI*, November 2006.

[Cor] EMC Corp. Symmetrix remote data facility. `http://www.emc.com/products/networking/srdf.jsp`. Last accessed April 2006.

[CPH01] C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In *Proc. of IFIP TC11 18th Int'l Conf. on Information Security and Privacy in the Age of Uncertainty (SEC)*, pages 73 – 84, November 2001.

[CV03] B. Chun and A. Vahdat. Workload and failure characterization on a large-scale federated testbed. Technical Report IRB-TR-03-040, Intel Research, November 2003.

[Dab05] F. Dabek. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, 2005.

[DF82] L. W. Dowdy and Derrell V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[DFM00] R. Dingledine, M. Freedman, and D. Molnar. The freehaven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[DG04] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of OSDI*, December 2004.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swindhart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of ACM PODC Symp.*, pages 1 – 12, 1987.

[DKK+01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[DLS+04] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proc. of NSDI*, March 2004.

[DR01] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[DW01] J. R. Douceur and R. P. Wattenhofer. Large-Scale Simulation of Replica Placement Algorithms for a Serverless Distributed File System. In *Proc. of MASCOTS*, 2001.

[DZD+03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, February 2003.

[EWK05] P. Eaton, H. Weatherspoon, and J. Kubiatowicz. Efficiently binding data to owners in distributed content-addressable storage systems. In *3rd International Security in Storage Workshop*, December 2005.

[Fet03] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, 2003.

[FKM00] K. Fu, M. F. Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proc. of OSDI*, October 2000.

[FLRS05] M. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2005.

[GBHC00] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. of OSDI*, October 2000.

[GCB+02] J. Gray, W. Chong, T. Barclay, A. Szalay, and J. vandenBerg. Terascale sneakernet: Using inexpensive disks for backup, archiving, and data exchange. Technical Report MSR-TR-2002-54, Microsoft Research, May 2002.

[GGL03] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of ACM SOSP*, pages 29–43, October 2003.

[GSK03] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[GWGR03] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Byzantine-tolerant erasure-coded storage. Technical Report CMU-CS-03-187, Carnegie Mellon University School for Computer Science, September 2003.

[HKRZ02] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proc. of ACM SPAA*, pages 41–52, August 2002.

[HMD05] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, May 2005.

[jBH+05] F. junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving internet catastrophe. In *Proc. of USENIX Annual Technical Conf.*, May 2005.

[K+00] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[KFM04] M. N. Krohn, M. J. Freedman, and D. Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *In Proc. of the IEEE Symp. on Security and Privacy*, pages 226 – 240, May 2004.

[KKM02] M. Karlsson, C Karamanolis, and M. Mahalingam. A framework for evaluationg replica placemement algorithms. Technical Report HPL-2002-219, Hewlett Packard Lab, 2002.

[KLL+97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees. distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, May 1997.

[KSL+04] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. D. Tygar. Distillation codes and their application to DoS resistant multicast authentication. In *Network and Distributed System Security Conference (NDSS 2004)*, pages 37–56, February 2004.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. of ACM SIGOPS*, 1991.

[LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of OSDI*, pages 121–136, December 2004.

[LMPZ02] S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proc. of USENIX FAST*, January 2002.

[LMS+97] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proc. of ACM STOC*, pages 150–159, 1997.

[LMS+98] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Analysis of low density codes and improved designs using irregular graphs. In *Proc. of ACM STOC*, May 1998.

[LS00] W. Litwin and T. Schwarz. LH* RS : A high-availability scalable distributed data structure using reed solomon codes. In *Proc. of ACM SIGMOD Conf.*, pages 237–248, May 2000.

[LSMK05] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of dht routing tables. In *Proc. of NSDI*, May 2005.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[LT96] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of ASPLOS*, pages 84–92, 1996.

[Lub02] M. Luby. Lt codes. In *Proc. of FOCS Symp.*, pages 271–282, November 2002.

[MA04] J-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, June 2004.

[May02] P. Maymounkov. Online codes. Technical Report TR2002-833, New York University, New York, NY, November 2002.

[Mer88] R. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378. Springer-Verlag, 1988.

[MGM04] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable dht data. Technical Report MIT-LCS-TR-993, MIT Laboratory for Computer Science, June 2004.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MMGC02] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[MPHD06] A. Mislove, A. Post, A. Haeberlen, and P. Druschel. Experiences in building and operating a reliable peer-to-peer application. In *Proc. of EuroSys Conf.*, April 2006.

[MR97] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of ACM STOC*, pages 569 – 578, May 1997.

[MRC+97] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, October 1997.

[MRG+05] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The lockss peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 23(1):2–50, 2005.

[MRTZ01] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *DISCEX II*, 2001.

[MT85] S. J. Mullender and A. S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proc. of ACM SOSP*, pages 51–62, December 1985.

[NIS94] NIST. FIPS 186 digital signature standard. `http://www.itl.nist.gov/fipspubs/fip186.htm`, May 1994.

[NYGS04] S. Nath, H. Yu, P.G. Gibbons, and S. Seshan. Tolerating correlated failures in wide-area monitoring services. Technical Report IRP-TR-04-09, Intel Research, May 2004.

[NYGS06] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proc. of NSDI*, May 2006.

[OP02] D. Oppenheimer and D. A. Patterson. Benchmarking large-scale internet services. In *Proc. of SIGOPS European Workshop*, September 2002.

[PCAR02] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, 2002.

[PFM06] L. Peterson, A. Bavier E. Fiuczynski, , and S. Muir. Experiences building planetlab. In *Proc. of OSDI*, November 2006.

[PGK88] D. Patterson, G. Gibson, and R. Katz. The case for RAID: Redundant arrays of inexpensive disks. In *Proc. of ACM SIGMOD Conf.*, pages 106–113, May 1988.

[PH02] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann, San Mateo, CA, May 2002.

[Pla97] J. Plank. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9):995–1012, September 1997.

[PMF+02] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File system based asynchronous mirroring for disaster recovery. In *Proc. of USENIX FAST*, January 2002.

[PP06] K. S. Park and V. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):65–74, January 2006. `http://comon.cs.princeton.edu/`.

[QD02] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proc. of USENIX FAST*, January 2002.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[REG+03] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, 2003.

[RGK+05] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, , and H. Yu. Opendht: A public dht service and its uses. In *Proc. of ACM SIGCOMM Conf.*, August 2005.

[RGRK04] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. of USENIX*, June 2004.

[RL03] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-932, MIT Laboratory for Computer Science, December 2003.

[RL05] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *Proc. of IPTPS*, March 2005.

[RP06] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *Proc. of INFOCOM*, April 2006.

[RS04] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of NSDI*, March 2004.

[RV97] L. Rizzo and L. Vicisano. A reliable multicast data distribution protocol based on software fec. In *Proc. of HPCS*, Greece, 1997.

[RWE+01] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SFV+04] Y. Saito, S. Frœlund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of ASPLOS*, pages 48–58, New York, NY, 2004. ACM Press.

[SHD+06] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and John Kubiatowicz. Proactive replication for data durability. In *Proc. of IPTPS*, Santa Barbara, CA, February 2006.

[Sho03] A. Shokrollahi. Raptor codes. Technical Report DF2003-06-01, Digital Fountain, Inc., Fremont, CA, June 2003.

[SK98] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. of USENIX Annual Technical Conf.*, January 1998.

[SLC+06] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Exploring the design of multi-site web services using the OverCite digital library. In *Proc. of NSDI*, San Jose, CA, May 2006.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM Conf.* ACM, August 2001.

[Str] Jeremy Stribling. Planetlab all-pairs ping. http://infospect.planet-lab.org/pings.

[Tvv+90] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[UoC] Berkeley University of California. Petabyte storage infrastructure project. http://elib.cs.berkeley.edu/ storage/psi.

[vS04] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, May 2004.

[WCSK05] H. Weatherspoon, B. Chun, C. W. So, and J. Kubiatowicz. Long-term data maintenance in wide-area storage systems: A quantitative approach. Technical Report UCB//CSD-05-1404, U. C. Berkeley, July 2005.

[WK02] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, March 2002.

[WMK02] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of Intl. Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[WWK02] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc. of Intl. Workshop on Future Directions of Distributed Systems*, June 2002.

[YNY+04] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[ZHS+04] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[ZJK01] B. Y. Zhao, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.

[ZSv00] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. Technical Report 2000-1828, Department of Computer Science, Cornell University, Ithaca, NY USA, 2000.

Appendix A

Durability Derivation

In this appendix we describe the mathematics involved in computing the *mean time to data loss* (MTTDL) of a *particular* erasure encoded block.

Considering the server failure model and repair process as described in Section 4.1.1, we can calculate the MTTDL of a block as follows. First, we calculate the probability that a given fragment placed on a randomly selected disk will survive until the next epoch as

$$p(e) = \int_e^\infty \frac{l p_d(l)}{\mu} \frac{l-e}{l} dl \quad (A.1)$$

$$= \frac{1}{\mu} \int_e^\infty p_d(l)(l-e) dl \quad (A.2)$$

where e is the length of an epoch, μ is the average life of a disk, and $p_d(l)$ is the probability distribution of disk lives. This equation is derived similarly to the equation for the residual average lifetime of a randomly selected disk. The term $\frac{l-e}{l}$ reflects the probability that, given a disk of lifetime l, a new fragment will land on the disk early enough in its lifetime to survive until the next epoch. The probability distribution $p_d(l)$ was obtained from disk failure distributions in [PH02], augmented by the assumption that all disks still in service after five years are discarded along with their data.

Next, given $p(e)$, we can compute the probability that a block can be reconstructed after a given epoch as

$$p_b(e) = \sum_{m=rn}^{n} \binom{n}{m} [p(e)]^m [1-p(e)]^{n-m} \quad (A.3)$$

where n is the number of fragments per block and r is the rate of encoding. This formula computes

the probability that at least *rn* fragments are still available at the end of the epoch.

Finally, the MTTDL of a block for a given epoch size can be computed as

$$\text{MTTDL}_{block}(e) = e \cdot \sum_{i=0}^{\infty} i[1 - p_b(e)][p_b(e)]^i \quad \text{(A.4)}$$

$$= e \cdot \frac{p_b(e)}{1 - p_b(e)}. \quad \text{(A.5)}$$

This last equation computes the average number of epochs a block is expected to survive times the length of an epoch.

Appendix B

Glossary of Terms

- **Availability:** See *data availability* or *server availability*.
- **Byzantine failure:** an arbitrary fault that occurs during the execution of a protocol. It encompasses those faults that are commonly referred to as ”crash failures” and ”send-receive omission failures”. When a Byzantine failure has occurred, the system may respond in any unpredictable way, unless it is designed to have Byzantine fault tolerance.

  These arbitrary failures are a superset of many failures and may be loosely categorized as follows:

  - a failure to take another step in the protocol, also known as a crash failure.
  - a failure to send or receive some message, also known as send-receive omission.
  - a failure to correctly execute a step of the protocol.
  - arbitrary execution of a step other than the one indicated by the protocol.

  Steps are taken by processes, the abstractions that execute the protocols. A faulty process is one that at some point exhibits one of the above failures. A process that is not faulty is correct.
- **Byzantine fault tolerance:** ability to defend against (or cope with) Byzantine failure and still satisfy the specification of a protocol. For instance, correctly functioning components of a Byzantine fault tolerant system will be able to reach the same group decision regardless of Byzantine faulty components. There are upper bounds on the percentage of unreliable components, however. Such algorithms are commonly characterized by their resilience, f,

the number of faulty processes with which an algorithm can cope. Many classic agreement problems, such as the Byzantine Generals Problem, have no solution unless $f < \frac{n}{3}$, where *n* is the number of processes in the system. See also *Byzantine failure*.

- **Byzantine process:** a faulty process that at some point exhibits one of the Byzantine failures. See also *Byzantine failure*.

- **Byzantine server:** See *Byzantine process*.

- **Checksum:** a digest or summary representation of a data object used to check whether errors have occurred during network transmission or in storage. The simplest form of checksum simply adds up the bits in the data object; however, such a scheme cannot detect a number of errors such as reordering of the bytes in a data object, inserting or deleting zero-valued bytes, and multiple errors which sum to zero. To provide protection against errors and security against malicious agents requires use of a *cryptographic hash* function.

- **Cryptographic Secure Checksum:** See *cryptographic secure hash function*.

- **Cryptographic Secure Hash:** a secure digest or summary representation of a data object used to check whether errors have occurred during network transmission or in storage. The additional security properties are collision resistance and uninvertibility. Collision-resistant means it should be hard to find two different data objects X and Y ($X! = Y$) such that hash H(X) = H(Y). Uninvertible means that given only the secure hash of a data object, $H(X)$ for example, it is computational infeasible to compute the data object, X. These security properties makes it difficult for error or a malicious attacker to corrupt data without detection.

- **Configuration:** Set of servers responsible for storing replicas for a particular data object.

- **Crash failure:** A process that either follows protocol correctly, produces correct results, or produces no results. For example, permanently failed server. This is the same as a fail-stop failure. See also *Byzantine server* and *crash server*.

- **Crash process:** a process that correctly follows specified protocol or permanently fails.

- **Crash-recovery failure:** A transient failure where a process eventually returns with state intact or a permanent failure where a process does not ever return. See also *crash-recovery process*, *permanent server failure*, and *transient server failure*.

- **Crash-recovery process:** A process that fails and *potentially* recovers (aka benign process). See also *crash-recovery failure, permanent server failure* and *transient server failure*.
- **Crash-recovery server:** See *crash-recovery process*.
- **Crash server:** See *crash process*.
- **Data:** An opaque sequence of bytes.
- **Data availability:** fraction of time a data object is available. The fraction of time a system can promptly return a requested data object.
- **Data durability:** probability that a data object exists after a specific amount of time. See also *data failure rate*.
- **Data failure rate:** number of times a particular (fixed-size) data object fails per unit time. For example, the fraction of blocks lost per year (FBLPY) is a failure rate. In the special case when the likelihood of failure remains constant as time passes such as with an exponential failure distribution, the failure rate is simply the inverse of the mean-time-to-failure (MTTF) for a particular data object. See also *data durability*.
- **Data fault tolerance:** ability to tolerate server failure without loss of data. It includes choosing the type of redundancy, number of replicas, and where to store replicas. See also *configuration, redundancy, data repair*, and *server failure*.
- **Data fault tolerance algorithm:** set of procedures used to define the components of an object's configuration. See also *configuration, data fault tolerance*;
- **Data fragment:** An original or encoded piece of a data object. Erasure-coding maps a data object broken into m original pieces (fragments) onto a larger set of n pieces ($n > m$) such that the original pieces can be recovered from a subset of all n pieces. The pieces that are not original are encoded. Since a piece may be as large as the whole data object, fragment and replica are often used interchangeably. See also *data replica, redundancy, replication*, and *erasure-coding*.
- **Data integrity:** ensures data stored into and returned from the storage system are the same.
- **Data repair:** the process of replacing replicas lost to server failure.

- **Data replica:** A whole copy of a data object. See also *data fragment*, *redundancy*, *replication*, and *erasure-coding*.
- **Disk:** A non-volatile storage substrate often directly attached to a server.
- **Downtime:** one contiguous interval of time when a server is unavailable. Commonly referred in the storage literature as a time-to-repair (TTR). See also *sessiontime* and *lifetime*.
- **Durability:** See *data durability*.
- **durable** means a data object exits. It is not possible to distinguish available from durable in a networked system limited to remote probing since data objects can exist but not be immediately available: if the only copy of a data object is on the disk of a server that is currently powered off, but will someday re-join the system with disk contents intact, then that data object exists but is not currently available. See also *data availability*.
- **Erasure-coding:** Erasure-coding maps a data object broken into m original fragments (pieces) onto a larger set of n fragments ($n > m$) such that the original fragments can be recovered from a subset of all n fragments. The fraction of the fragments required is called the rate, denoted r. Optimal erasure codes such as Reed-Solomon [BKK+95, Pla97, RV97] codes produce $n = m/r$ ($r < 1$) fragments where any m fragments are sufficient to recover the original data object. Unfortunately optimal codes are costly (in terms of memory usage, CPU time or both) when m is large, so near optimal erasure codes such as Tornado codes [LMS+97, LMS+98] are often used. These require $(1+\varepsilon)m$ fragments to recover the data object. Reducing ε can be done at the cost of CPU time. Alternatively, rateless erasure codes such as LT [Lub02], Online [May02], or Raptor [Sho03], codes transform a data object of m fragments into a practically infinite encoded form. Data loss occurs a sufficient fraction of fragments are lost due to permanent server failure. Erasure-codes often provide redundancy without storage overhead of strict replication. See also *redundancy*, *rate of encoding*, *storage overhead*.
- **Failure:** See *permanent server failure* and *transient server failure*.
- **Fault tolerance:** See *data fault tolerance*.
- **Fault tolerance algorithm:** See *data fault tolerance algorithm*.
- **Fail-stop:** A process that either follows protocol correctly, produces correct results, or produces no results. For example, permanently failed server. This is the same as a crash failure.

- **Failure rate:** See *data failure rate*.
- **Fragment:** See *data fragment*.
- **Immutable:** Cannot change. Read-only. For example, an immutable data object is read-only and cannot change. See also *mutable*.
- **Integrity:** state of data stored in the system always reflects changes made by the owner. It includes non-repudiably binding owner to data and ensuring that only the owner can modify data, returned data is the same as stored, and the last write is returned in subsequent reads. See also *non-repudiation*, *data integrity*, and *order integrity*.
- **Lifetime:** time between when a component first enters and last leaves a system. In terms of sessiontime it is the time between the beginning of the first session and end of last session. For example, a server's lifetime is comprised of a number of interchanging session and downtimes.
- **Mean-time-between-failure (MTBF):** the average time between failures, the reciprocal of the failure rate in the special case when failure rate is constant. Calculations of MTBF assume that a system is "renewed", i.e. fixed, after each failure, and then returned to service immediately after failure.
- **Mean-time-to-failure (MTTF):** average sessiontime.
- **Mean-time-to-repair (MTTR):** average downtime.
- **Memory:** A volatile storage substrate often directly attached to a server.
- **Mutable:** subject to change or alteration. For example, a mutable data object can change. A mutable data object is readable and writable data object. See also *immutable*.
- **Node:** a basic unit used to build systems. For example, in a distributed system, a server is a particular type of node. A server satisfies remote requests and/or participates a in peer-to-peer network. See also *server*.
- **Non-repudiation:** Cannot deny
- **Object:** See *data*.
- **Object availability:** See *data availability*.

- **Object durability:** See *data durability*.
- **Order integrity:** defines a single sequence of writes where each write has a unique sequence number and writes can be ordered based on sequence number (aka total order). For example, in the absence of new writes order integrity often ensures data returned from a storage system was the most recently written data.
- **Permanent failure:** See *permanent server failure*.
- **Permanent data loss:** data can no longer be retrieved or reconstructed from information within the system. See also *durability*.
- **Permanent server failure:** loss of data stored by a server. Examples include disk crash, server reinstallation or departure from network without return. Permanent server failure results in loss of redundancy durably stored. See *data durability* and *transient server failure*.
- **Rate of encoding:** size of the original data object divided by the encoded size. For example, the ratio between the number of fragments required to reconstruct the whole data object and the redundancy (total number of original and encoded fragments).
- **Redundancy:** duplication of data in order to reduce the risk of permanent data loss. The total number of whole copies or unique pieces of data. See also *replication* and *erasure coding*.
- **Replica Location and Repair Service:** A service used to locate and monitor data object replicas and trigger a repair process when necessary. See also *data repair*.
- **Repair:** See *data repair*.
- **Replica:** See *data replica*.
- **Replication:** duplication of data to reduce the risk of permanent data loss via creating whole, identical, copies of data. See also *redundancy* and *erasure-coding*.
- **Server:** a node that satisfies remote requests and/or participates in a peer-to-peer network. A server often is a computer with processor(s), memory, small number of disk drives, and a set of networking ports.
- **Server Availability:** The percent of time a server is capable of responding to requests. The cumulative sessiontime divided by lifetime is a common measure of server availability, which is equivalent to the more commonly known expression in the storage literature $\frac{MTTF}{MTTF+MTTR}$.

- **Server failure:** See *permanent server failure* and *transient server failure*.
- **Sessiontime:** one contiguous interval of time when a server is available. Commonly referred in the storage literature as a time-to-failure (TTF). See also *downtime* and *lifetime*.
- **Storage overhead:** The total number of whole copies. Or the ratio between the redundancy (total number of original and encoded fragments) and the number of fragments required to reconstruct the whole data object.
- **Time-to-failure (TTF):** See *sessiontime*.
- **Time-to-repair (TTR):** See *downtime*.
- **Transient failure:** See *transient server failure*.
- **Transient server failure:** Loss of server availability. Examples include server reboot, network and power outage, and software crash where server returns from failure with data intact. Transient server failure does not decrease data durability; however, it does cause data to become unavailable. Transient server failure does not (conceptually) affect systems primarily concerned with data durability; however, it is not possible for systems to perfectly distinguish transient from permanent server failure.